

x86汇编语言

从实模式到保护模式

李忠 王晓波 余洁◎著

- 你想知道汇编语言到底是什么吗？
- 你想探究x86处理器的内部原理吗？
- 你想从侧面理解操作系统的工作原理，并尝试打造一个自己的系统吗？
- 这本书将会带你学习和掌握汇编语言程序设计的基本方法，了解汇编语言和处理器、计算机系统以及操作系统之间的关系。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY



x86汇编语言

从实模式到保护模式

李忠 王晓波 余洁◎著

- 你想知道汇编语言到底是什么吗？
- 你想探究x86处理器的内部原理吗？
- 你想从侧面理解操作系统的工作原理，并尝试打造一个自己的系统吗？
- 这本书将会带你学习和掌握汇编语言程序设计的基本方法，了解汇编语言和处理器、计算机系统以及操作系统之间的关系。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

x86汇编语言从实模式到保护模式

李　忠　王晓波　余　洁　著

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内容简介

本书采用开源的NASM 汇编语言编译器和VirtualBox 虚拟机软件，以个人计算机广泛采用的Intel处理器为基础，详细讲解了Intel 处理器的指令系统和工作模式，以大量的代码演示了16／32／64 位软件的开发方法，集中介绍处理器的16 位实模式和32 位保护模式，以及基本的指令系统。

这是一本有趣的书，它没有把篇幅花在计算一些枯燥的数学题上。相反，它教你如何直接控制硬件，在不借助于BIOS、DOS、Windows、LINUX 或任何其他软件支持的情况下来显示字符、读取硬盘数据、控制其他硬件等。本书可作为大专院校相关专业学生和计算机编程爱好者的教程。

图书在版编目（CIP）数据

x86 汇编语言：从实模式到保护模式 / 李忠，王晓波，余洁著. —北京：电子工业出版社，2013.1

ISBN 978-7-121-18799-5

Ⅰ.①x… Ⅱ.①李…②王…③余… Ⅲ.①汇编语言—程序设计 Ⅳ.①TP313

中国版本图书馆CIP 数据核字（2012）第253290 号

责任编辑：董亚峰 特约编辑：王 纲

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编 100036

开 本：787×1092 1/16 印张：24.25 字数：620千字

印 次：2013年1月第1次印刷

定 价：56.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

尽管汇编语言也是一种计算机语言，但却是与众不同的，与它的同类们格格不入。处理器的工作是执行指令并获得结果，而为了驾驭处理器，汇编语言为每一种指令提供了简单好记、易于书写的符号化表示形式。

一直以来，人们对于汇编语言的认识和评价可以分为两种，一种是觉得它非常简单，另一种是觉得它学习起来非常困难。

你认为我会赞同哪一种？说汇编语言难学，这没有道理。学习任何一门计算机语言，都需要一些数制和数制转换的知识，也需要大体上懂得计算机是怎么运作的。在这个前提下，汇编语言是最贴近硬件实体的，也是最自然和最朴素的。最朴素的东西反而最难掌握，这实在说不过去。因此，原因很可能出在我们的教科书上，那些一上来就搞一大堆寻址方式的书，往往以最快的速度打败了本来激情高昂的初学者。

但是，说汇编语言好学，也同样有些荒谬。据我的观察，很多人掌握了若干计算机指令，会编写一个从键盘输入数据，然后进行加减乘除或者归类排序的程序后，就认为自己掌握了汇编语言。还有，直到现在，我还经常在网上看到学生们使用DOS 中断编写程序，他们讨论的也大多是实模式，而非32 位或者64 位保护模式。他们知道如何编译源程序，也知道在命令行输入文件名，程序就能运行了；又或者使用一个中断，就能显示字符。至于这期间发生了什么，程序是如何加载到内存中的，又是怎么重定位的，似乎从来不关汇编语言的事。这样做的结果，就是让人以为汇编语言不过如此，而且非常枯燥。

很难说我已经掌握了汇编语言的要义。但至少我知道，尽管汇编语言不适合用来编写大型程序，但它对于理解计算机原理很有帮助，特别是处理器的工作原理和运行机制。就算是为了这个目的，也应该让汇编语言回归它的本位，那就是访问和控制硬件（包括处理器），而不仅仅是编写程序，输入几个数字，找出正数有几个、负数有几个，大于30 的有几个。

事实上，汇编语言对学习和理解高级语言，比如C 语言，也有极大的帮助。老教授琢磨了好几天，终于想到一个好的比喻来帮助学生理解什么是指针，实际上，这对于懂得汇编语言的学生来说，根本就不算个事儿，并因此能够使老教授省下时间来喝茶。

在这本书之前，我也写过《穿越计算机的迷雾》一书。它们是一个系列，没有基础的读者可以先看那本书，打一点计算机原理的基础再来学习汇编语言。

在计划写这本书的时候，我就给自己画了几条线。首先不能走老路，一上来就讲指令、寻址方式，而是采用任务驱动的方式来写，每一章都要做点事情，最好是比较有趣，有吸引力。在解决问题的过程中，不断地引入新指令，并进行讲解。一句话，我希望是润物细无声式的；其次，汇编语言和硬件并举，完全抛弃BIOS 中断和DOS 中断，直接访问硬件，发挥汇编语言的长处，因为这才是我们学习汇编语言的目的。也只有这样，读者才能深刻体会到汇编语言的妙处。

王晓波和湖北经济学院的余洁共同参与了本书的创作。

本书主要讲述INTEL x86 处理器的16 位实模式、32 位保护模式，至于虚拟8086 模式，则是为了兼容传统的8086 程序，现在看来已经完全过时，不再进行讲述。本书的特色之一是提供了大量典型的源代码，这些代码以及相配套的工具程序可以到书中指定的网站，或者电子工业出版社华信教育资源网搜索下载。

很多读者在读书的时候会遇到这种情况：一开始读得很快，一口气读了好几章。随着内容的深入，学习越来越吃力，不得不频繁回到前面重新学习已经讲过的内容，这就是因为前面的知识没有完全掌握。为此，本书每一章都设有检测点，读者应当在通过检测点之后再继续往后阅读。

本书原来有18 章，后来考虑到实模式的内容过多，而去掉了一章。这一章的标题是《聆听数字的声音》，讲述如何通过直接访问和控制Sound Blaster 16 声卡来播放声音，感兴趣的朋友可以从下载的配书文件包中找到这部分内容。

在本书的写作和出版过程中，长春电视台台长王志强，副台长周武军和技术部主任刘贵先后对本书给予了关心和支持，在此表示衷心

的感谢。

好友王南洋、桑国伟、刘维钊、蒋胜友、邱海龙、万利、李文心等负责了本书的一部分校对工作；好友周卫平帮我验证配书代码是否能在他的机器上正常工作，在这里向他们表示感谢，同时也谢谢所有关心和支持本书的朋友们。

感谢我的母亲、我的妻子和我的女儿，她们是我的精神支柱，是我努力创作这本书的动力来源。

在阅读本书的过程中，如果有任何问题，可以往电子邮件地址leechung@126.com 给我写信；要了解其他更多的情况，请访问我的博客：http://blog.163.com/leechung@126。

二〇一二年十一月

目　录

第1部分　预备知识

了解数制的基本知识和数制转换的方法。

了解8086处理器的结构和工作方式，初步认识所谓的针对处理器编程，是针对处理器的哪些部件和哪些方面进行的，理解分段的原理。

了解什么是汇编语言，以及如何书写、编译汇编语言源程序，掌握在虚拟机上运行程序的方法。

第1章　十六进制计数法

电子计算机，顾名思义，就是计算的机器。因此，学习汇编语言，就不可避免地要和数字打交道。在这个过程中，我们要用到三种数制：十进制（这是我们再熟悉不过的）、二进制和十六进制。本章的目标是：

1．熟悉后两种数制，了解这两种数制的计数特点。

2．能够在这三种数制之间熟练地进行转换，特别是看到一个二进制数时，能够口算出它对应的十六进制数，反之亦然。

3．对于0～15 之间的任何一个十进制数，能够立即说出它对应的二进制数和十六进制数。

1.1 二进制计数法回顾

1.1.1 关于二进制计数法

在《穿越计算机的迷雾》那本书里我们已经知道，计算机也是一台机器，唯一不同的地方在于它能计算数学题，且具有逻辑判断能力。

与此同时，我们也已经在那本书里学到，机器在做数学题的时候，也面临着一个如何表示数字的问题，比如你采用什么办法来将加数和被加数送到机器里。

同样是在那本书里，我们揭晓了答案，那就是用高、低两种电平的组合来表示数字。如图1-1所示，参与计算的数字通过电线送往计算机器，高电平被认为是“1”，低电平被认为是“0”，这样就形成了一个序列“11111010”，这就是一个二进制数，在数值上等于我们所熟知的二百五，换句话说，等于十进制数250。

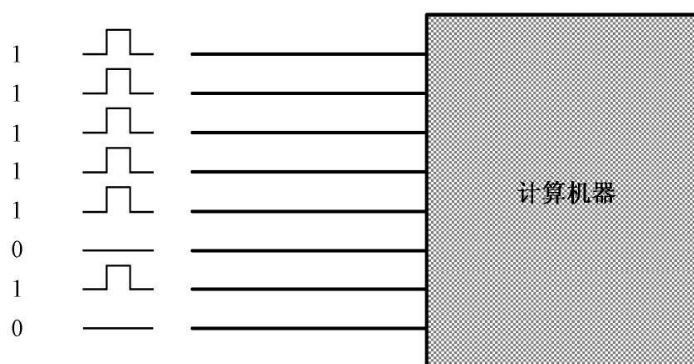


图1-1　在计算机里，二进制数字对应着高低电平的组合

从数学的角度来看，二进制计数法是现代主流计算机的基础。一方面，它简化了硬件设计，因为它只有两个符号“0”和“1”，要得到它们，可以用最少的电路元件来接通或者关断电路就行了；另一方面，二进制数与我们熟悉的十进制数之间有着一对一的关系，任何一个十进制数都对应着一个二进制数，不管它有多大。比如，十进制数5，它所对应的二进制数是101，而十进制数5785478965147则对应着一长串“0”和“1”的组合，即1010100001100001001011010110010011110011011。

组成二进制数的每一个数位，称为一个比特（bit），而一个二进制数也可以看成是一个比特串。很明显，它的数值越大，这个比特串就越长，这是二进制计数法不好的一面。

1.1.2　二进制到十进制的转换

每一种计数法都有自己的符号（数符）。比如，十进制有0、1、2、3、4、5、6、7、8、9 这十个符号；二进制呢，则只有0、1 这两个符号。这些数字符号的个数称为基数。也就是说，十进制有10 个基数，而二进制只有两个。

二进制和十进制都是进位计数法。进位计数法的一个特点是，符号的值和它在这个数中所处的位置有关。比如十进制数356，数字6 处在个位上，所以是“6 个”；5 处在十位上，所以是“50”；3 处在百位上，所以是“300”。即：

百位3、十位5、个位6＝$3\times10^2+5\times10^1+6\times10^0=356$

这就是说，由于所处的位置不同，每个数位都有一个不同的放大倍数，这称为“权”。每个数位的权是这样计算的（这里仅讨论整数）：从右往左开始，以基数为底，指数从0 开始递增的幂。正如上面的公式所清楚表明的那样，“6”在最右边，所以它的权是以10 为底，指数为0 的幂10^0；而3 呢，它的权则是以10 为底，指数为2 的幂10^2 。

上面的算式是把十进制数“翻译”成十进制数。从十进制数又算回到十进制数，这看起来有些可笑，注意这个公式是可以推广的，可以用它来将二进制数转换成十进制数。

比如一个二进制数10110001，它的基数是2，所以要这样来计算与它等值的十进制数：

$$10110001B=1\times2^7+0\times2^6+1\times2^5+1\times2^4+0\times2^3+0\times2^2+0\times2^1+1\times2^0=177D$$

在上面的公式里，10110001B 里的“B”表示这是一个二进制数，“D”则表示177 是个十进制数。“B”和“D”分别是英语单词Binary 和Decimal 的头一个字母，这两个单词分别表示二进位和十进位的意思。

检测点1.1

将下列二进制数转换成十进制数：

1101、1111、1001110、11111111、10000000、1101101100011011

1.1.3　十进制到二进制的转换

为了将一个十进制数转换成二进制数，可以采用将它不停地除以二进制的基数2，直到商为0，然后将每一步得到的余数串起来即可。如图1-2 所示，如果要将十进制数26 转换成二进制数11010，那么可采用如下方法：

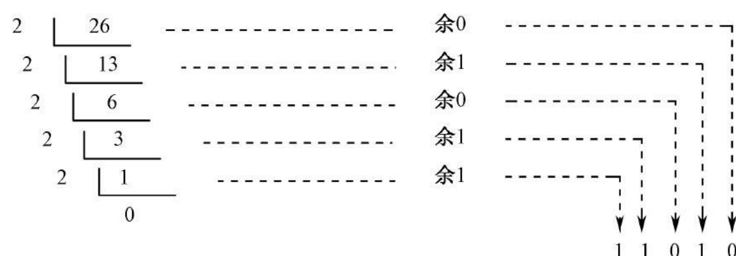


图1-2　将十进制数26 转换成二进制数

第1 步，将26 除以2，商为13，余数为0；

第2 步，用13 除以2，商为6，余数为1；

第3 步，用6 除以2，商为3，余数为0；

第4 步，用3 除以2，商为1，余数为1；

第5 步，用1 除以2，商为0，余数为1，结束。

然后，从下往上，将每一步得到的余数串起来，从左往右书写，就是我们所要转换的二进制数。

检测点1.2

将下列十进制数转换成二进制数：

8、10、12、15、25、64、100、255、1000、65535、1048576

1.2　十六进制计数法

1.2.1　十六进制计数法的原理

二进制数和计算机电路有着近乎直观的联系。电路的状态，可以用二进制数来直观地描述，而一个二进制数，也容易使我们仿佛观察到了每根电线上的电平变化。所以，我们才形象地说，二进制是计算机的官方语言。

即使是在平时的学习和研究中，使用二进制也是必需的。一个数字电路输入什么，输出什么，电路的状态变了，是哪一位发生了变化，研究这些，肯定要精确到每一个比特。这个时候，采用二进制是最直观的。

但是，二进制也有它的缺点。眼下看来，它最主要的缺点就是写起来太长，一点也不方便。为此，人们发明了十六进制计数法。至于为什么要发明另外一套计数方法，而不是依旧采用我们熟悉的十进制，下面就要为大家解释。

一旦知道二进制有两个数符“0”和“1”，十进制有十个数符“0”到“9”，那么我们就会很自然地认为十六进制一定有16 个数符。

一点没错，完全正确。这16 个数符分别是0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F。

你可能会觉得惊讶，字母怎么可以当做数字来用？这样的话，那些熟悉的英语单词，像Face（脸）、Bad（坏的）、Bed（床）就都成了数。

这又有什么奇怪的？你觉得“0”、“5”、“9”是数字，而“A”、“B”不是数字，这是因为你已经从小习惯了这种做法。

对于自然数里的前10 个，十进制和十六进制的表示方法是一致的。但是，9 之后的数，两者的表示方法就大相径庭了，如表1-1 所示。

表1-1　部分十进制数和十六进制数对照表

十进制数	十六进制数	十进制数	十六进制数
0	0	17	11
1	1	18	12
2	2	19	13
3	3	20	14
4	4	21	15
5	5	22	16
6	6	23	17
7	7	24	18
8	8	25	19
9	9	26	1A
10	A	27	1B
11	B	28	1C

续表

十进制数	十六进制数	十进制数	十六进制数
12	C	29	1D
13	D	30	1E
14	E	31	1F
15	F	32	20
16	10	33	21

很显然，一旦某个数位增加到9 之后，下一次，它将变成A，而不是向前进位，因为这里是逢16 才进位的。进位只发生在某个数位原先是F的情况下，比如1F，它加一后将会变成20。

1.2.2　十六进制到十进制的转换

要把一个十六进制数转换成我们熟悉的十进制数，可以采用和前面一样的方法。只不过，计算各个数位的权时，幂的底数是16。比如将十六进制数125 转换成十进制数的方法如下：

$$125H=1\times16^2+2\times16^1+5\times16^0=293D$$

上式里，125 后面的“H”用于表明这是一个十六进制数，它是英语单词Hexadecimal 的头一个字母，这个单词的意思是十六进制。

检测点1.3

将下列十六进制数转换成十进制数：

8、A、B、C、D、E、F、10、1F、6CD、3FE、FFC、FFFF

1.2.3　十进制到十六进制的转换

如图1-3 所示，相应地，要把一个十进制数转换成十六进制数，则可以采取不停地除以16 并取其余数的策略。

第1 次，将293 除以16，商为18，余5；

第2 次，用18 除以16，商为1，余2；

第3 次，再用1 除以16，商为0，余1，结束。

然后，从下往上，将每次的余数1、2、5 列出来，得到125，这就是所要的结果。

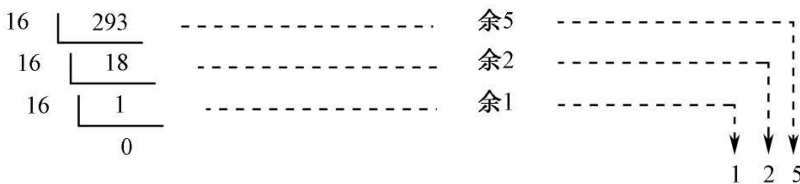


图1-3　将十进制数293 转换成十六进制数

检测点1.4

将下列十进制数转换成十六进制数：

8、10、12、15、25、64、100、255、1000、65535、1048576

1.2.4　为什么需要十六进制

为什么我们要发明十六进制计数法？为什么我们要学习它？

提出这样的问题，在我看来很有趣，也很有意义，但似乎从来没有人在书上正面回答过。这样一来，可怜的学子们只能在掌握了十六进制若干年之后，在某一天里自己恍然大悟。

为了搞清楚这个问题，我们不妨来列张表（表1-2），看看十进制数、二进制数和十六进制数之间，都有些什么有趣的规律和特点。

表1-2　部分十进制数、二进制数和十六进制数对照表

十进制数	二进制数	十六进制数	十进制数	二进制数	十六进制数
0	0000	0	10	1010	A
1	0001	1	11	1011	B
2	0010	2	12	1100	C
3	0011	3	13	1101	D
4	0100	4	14	1110	E
5	0101	5	15	1111	F
6	0110	6	16	0001 0000	10
7	0111	7	17	0001 0001	11
8	1000	8	55	0011 0111	37
9	1001	9	195	1100 0011	C3

在上面这张表里（表1-2），每一个二进制数在排版的时候，都经过了“艺术加工”，全都以4比特为一组的形式出现。不足4 比特的，前面都额外加了“0”，比如10，被写成0010 的形式。就像十进制数一样，在一个二进制数的前面加多少个零，都不会改变它的值。

注意观察这张表并开动脑子，4 比特的二进制数，可以表示的数是0000 到1111，也就是十进制的0～15，这正好对应于十六进制的0～F。

在这个时候，如果将它们都各自加1，那么，下一个二进制数是0001 0000，与此同时，它对应的十六进制数则是10，你会发现，它们有着如图1-4（左边）所示的奇妙对应关系。



图1-4　十六进制的每一位与二进制数每4 比特为一组的对应关系

再比如图1-4（右边）中的二进制数1100 0011，它与等值的十六进制数C3 也有着相同的对应关系。

也就是说，如果将一个二进制数从右往左，分成4 比特为一组的形式，分别将每一组的值转换成十六进制数，就可以得到这个二进制数所对应的十六进制数。

这样一来，如果我们稍加努力，将0～F 这16 个数所对应的二进制数背熟，并能换算自如的话，那么，当我们看到一个十六进制数3F8 时，

我们就知道，因为3 对应的二进制数为0011，F对应的二进制数是1111，8 对应的二进制数是1000，所以3F8H＝0011 1111 1000B。

同理，如果一个二进制数是1101 0010 0101 0001，那么，将它们按4 比特为一组，分别换算成十六进制数，就得到了D251。

正如前面所说的，从事计算机的学习和研究（包括咱们马上就要进行的汇编语言程序设计），不可避免地要与二进制数打交道，而且有时还必须针对其中某些比特进行特殊处理。这个时候，如果想保留二进制数的直观性，同时还要求写起来简短，十六进制数是最好的选择。

检测点1.5

1．将下列十六进制数转换成二进制数：

3、A、C、F、20、3F、2FE、FFFF、9FC05D、7CCFFEFF

2．快速说出以下十进制数所对应的二进制数和十六进制数：

1、3、5、7、9、11、13、15、0、2、4、6、8、10、12、14

1.3　使用Windows 计算器方便你的学习过程

和十进制数一样，二进制数和十六进制数也可以进行加、减、乘、除运算。比如，两个十六进制数F 和A 相乘，结果是十六进制数96。从十进制的角度来看这个计算过程，就是两个十进制数15 和10 相乘，结果为150。

在学习汇编语言程序设计的过程中，出于解决实际问题的需要，经常要在编写程序时做一些计算工作。十进制就不说了，我们都很熟悉，计算起来驾轻就熟。但是，如果是几个二进制数进行加减乘除，或者几个十六进制数加减乘除，就很困难了。想想看，为了做十进制乘法，我们要背九九乘法口诀。而十六进制有16 个基数，它的乘法口诀就更多了。

这本书的目的不是教会你十六进制四则运算的方法和步骤，不是这样的。相反，我希望你能借助于一些工具来快速得到计算结果，从而把更多的精力放到学习汇编语言上。

不是所有知识都应当放在脑子里，要善于利用工具！

为了将较大的数转换成不同的数制，或者进行某种数制的四则运算，可以使用Windows 计算器。这是一个小软件，每个版本的Windows 操作系统都有，你应该很熟悉，其界面如图1-5 所示。注意，如果该程序运行后的界面与此不同，则可以通过选择菜单“查看”->“程序员”进行更改。



图1-5　Windows 计算器

计算器软件的使用方法并不复杂，只需稍加练习即可掌握。比如，选择单选钮“十六进制”，然后输入一个十六进制数。此时，如果你再选择单选钮“十进制”，则刚才输入的内容就会立即变成十进制的形式，这就是进行数制转换的一个例子。

检测点1.6

1．用计算器程序将FFCH 转换成十进制数和二进制数；

2．用计算器程序计算FFCH 乘以27C0H 的结果，并转换成二进制数。

本章习题

1. 口算：

5H=___D 12D=___H 0FH=___D=___B

0CH=___D=___B 0AH=___D=___B 8D=___H=___B

0BH=___D=___B 0EH=___D=___B 10H=___D=___B

2. 口算：

10010B=___H 15H=___B 8FH=___B 200H=___B 111111111B=___H

第2章　处理器、内存和指令

鉴于汇编语言和处理器之间的紧密关系，学习汇编语言的过程，实际上也是洞悉处理器内部构造和工作方式的过程。在本章中，我们要借助于一款早已淘汰的处理器INTEL8086 来了解处理器、内存和指令这三者之间的关系。不要小看这款处理器，它是整个INTEL x86 处理器家族的起点和基础。本章的目标是：

1．了解INTEL8086 处理器的通用寄存器和段地址加偏移地址的内存访问方式。

2．了解分段机制对程序重定位的好处。

3．理解INTEL8086 处理器内存分段的本质，充分认识到这种分段机制的灵活性。

2.1 最早的处理器

1947 年，美国贝尔实验室的肖克利和同事们一起发明了晶体管。1958 年，也许是受够了在一大堆晶体管里连接那些杂乱无章的导线，另一个美国人杰克·基尔比发明了集成电路。接着，1971 年，在为日本人设计计算器芯片的过程中，受到启发的Intel 公司生产了世界上第一个处理器INTEL 4004。

图2-1 所示的是INTEL 4004 处理器和它的设计者弗德里科·法金（Federico Faggin）。

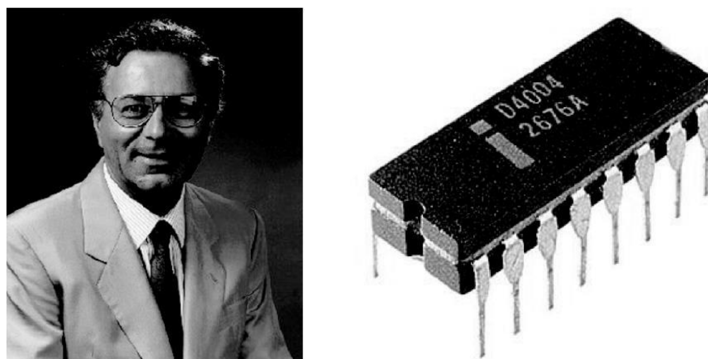


图2-1 Intel 第一块处理器4004 和它的设计者弗德里科·法金

我们已经知道，处理器（Processor）是一台电子计算机的核心，它会在振荡器脉冲的激励下，从内存中获取指令，并发起一系列由该指令所定义的操作。当这些操作结束后，它接着再取下一条指令。通常情况下，这个过程是连续不断、循环往复的。

2.2　寄存器和算术逻辑部件

为什么处理器能够自动计算，这个问题已经在我的上一本书《穿越计算机的迷雾》里讲过了，不过这些原理讲起来很费劲，花了整整一本书的篇幅。当然，如果你没看过这本书，也没关系，下面就来简单回顾一下。回顾这些知识很有用，因为只有这样你才能知道如何安排处理器做事情。

电子计算机能做很多事情。你能够知道明天出门要穿厚一点才不挨冻，是因为电子计算机算出了天气。除此之外，它还能让你看电影、听音乐、写文章、上网。尽管表面上看来，这些用处和算数学题没什么关系，但实质上，这些功能都是以数学计算为基础的。正是因为如此，人们才会把“计算”这个词挂在嘴边，什么“云计算”、“网络计算”、“64 位计算”，等等。

处理器不是法师手里的仙器，它之所以能计算数学题，是因为其特殊的设计。处理器是一个“器”，即器件，不太大，有的是长方形，有的是正方形，就像饼干。实际上，它是一块集成电路。

如图2-2 所示，在处理器的底部或者四周，有大量的引脚，可以接受从外面来的电信号，或者向外发出电信号。每个引脚都有自己的用处，在往电路板上安装的时候，不能接错。所以，如图中所示，处理器在生产的时候，都会故意缺一个角，这是一个参照标志，可以确保安装的人不会弄错。当然，并不是所有的处理器都会缺一个角，这不是一个固定不变的做法。

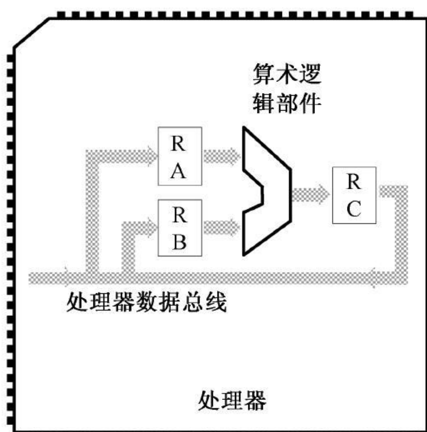


图2-2　处理器进行数学运算的简单原

处理器的引脚很多，其中有一部分是用来将参与运算的数字送入处理器内部。有些引脚是复用的，假如现在要进行加法运算，那么我们要重复使用这些引脚，来依次将被加数和加数送入。

一旦被加数通过引脚送入处理器，代表这个二进制数字的一组电信号就会出现在与引脚相连的内部线路上。这是一排高低电平的组合，代表着二进制数中的每一位。这时候，必须用一个称为寄存器（Register）的电路锁住。之所以要这样做，是因为相同的引脚和线路马上还要用于输入加数。也正是因为这个原因，这些内部线路称为处理器内部总线。

同样地，加数也要锁进另一个寄存器中。如图2-2 所示，寄存器RA 和RB 将分别锁存参与运算的被加数和加数。此后，RA 和RB 中的内容不再受外部数据线的影响。

寄存器是双向器件，可以在一端接受输入并加以锁存，同时，它也会在另一端产生一模一样的输出。与寄存器RA 和RB 相连的，是算术逻辑单元，或者算术逻辑部件（Arithmetic Logic Unit，ALU），也就是图2-2 中的桶形部分。它是专门负责运算的电路，可以计算加法、减法或者乘法，也可做逻辑运算。在这里，我们要求它做一次加法。

一旦寄存器RA 和RB 锁存了参与运算的两个数，算术逻辑部件就会输出相加的结果，这个结果可以临时用另外一个寄存器RC 锁存，稍后再通过处理器数据总线送到处理器外面，或者再次送入RA 或RB。

处理器内部有一个控制器（图中没有画出），在指令的执行过程中，它负责给各个部件发送控制信号，使各个部件在某个正确的时间点上执行某个动作。同时，它还负责决定在某个时间点上哪个部件有权使用总线，以免彼此发生冲突。

处理器总是很繁忙的，在它操作的过程中，所有数据在寄存器里面都只能是临时存在一会儿，然后再被送往别处，这就是为什么它被叫做“寄存器”的原因。早期的处理器，它的寄存器只能保存4 比特、8 比特或16 比特，分别叫做4 位、8 位和16 位寄存器。现在的处理器，寄存器一般都是32 位、64 位甚至更多。

如图2-3 所示，8 位寄存器可以容纳8 比特（bit），或者说1 字节（Byte），这是因为

$$1\ \text{byte} = 8\ \text{bit}$$

另外，我们还要为这个字节的每一位编上号，编号是从右往左进行的，从0 开始，分别是0、1、2、3、4、5、6、7。在这里，位0（第1 位）是最低位，在最右边；位7（第8 位）是最高位，在最左边。

为了更好地理解上面这些概念，图中假定8 位寄存器里存放的是二进制数10001101，即十六进制的8D。这时，它的最低位和最高位都是1。

16 位寄存器可以存放2 个字节，这称为1 个字（word），各个数位的编号分别是0～15，其中0～7 是低字节，8～15 是高字节。实际上，“字”的概念出现得很早，也并非指16 个比特。只是到了后来，才特指16 个二进制位的长度。

32 位寄存器可以存放4 个字节，这称为1 个双字（double word），各个数位的编号分别是0～31，其中0～15 是低字，16～31 是高字。

尽管图中没有画出，但是64 位寄存器可以容纳更多的比特，也就是8 个字节，或者4 个字。位数越多，寄存器所能保存的数越大，这是显而易见的。

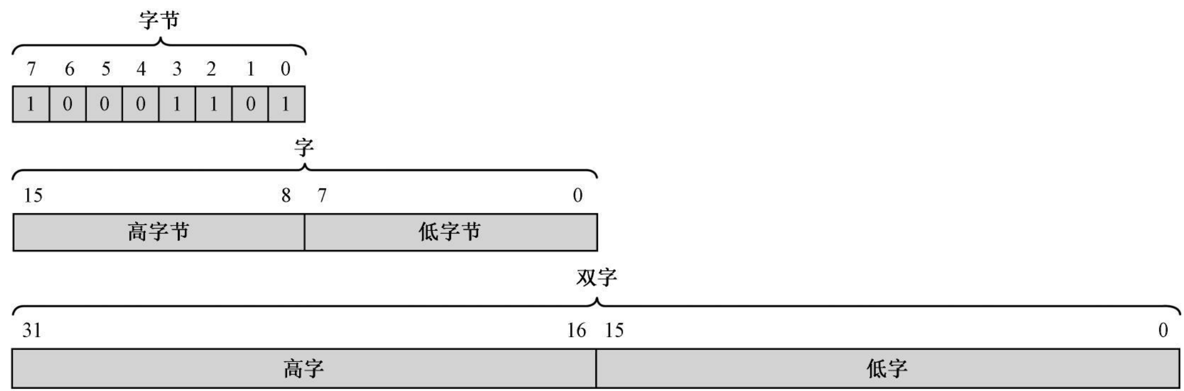


图2-3　寄存器数据宽度示意

2.3 内存储器

前面已经讲过，处理器的计算过程，实际上是借助于寄存器和算术逻辑部件进行的。那么，参与计算的数是从哪里来的呢？答案是一个可以保存很多数字的电路，叫做存储器（Storage 或 Memory）。

存储器的种类实际上是很多的，包括大家都知道的硬盘和U 盘等。甚至寄存器就是存储器的一种。不过，我们现在所要讲到的存储器，则是另外一种东西。

如图2-4 所示，这是所有个人计算机里都会用到的存储器，我们平时把它叫做内存条。这个概念是这么来的，首先，它是计算机内部最主要的存储器，通常只和处理器相连，所以叫做内存储器或者主存储器，简称内存或主存。其次，它一般被设计成扁平的条状电路板，所以叫内存条。

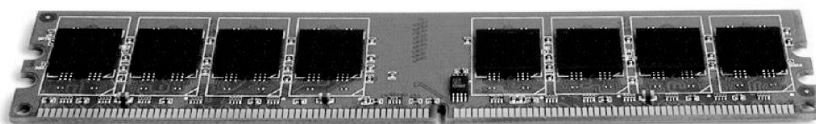


图2-4　个人计算机里使用的内存条

如图2-5 所示，和寄存器不同，内存用于保存更多的比特。对于用得最多的个人计算机来说，内存按字节来组织，单次访问的最小单位是1 字节，这是最基本的存储单元。如图中所示，每个存储单元中，各位的编号分别是0～7。

内存中的每字节都对应着一个地址，如图2-5 所示，第1 个字节的地址是0000H，第2 个字节的地址是0001H，第3 个字节的地址是0002H，其他以此类推。注意，这里采用的是十六进制表示法。作为一个例子，因为这个内存的容量是65536 字节，所以最后一个字节的地址是FFFFH。

为了访问内存，处理器需要给出一个地址。访问包括读和写，为此，处理器还要指明，本次访问是读访问还是写访问。如果是写访问，则还要给出待写入的数据。

8 位处理器包含8 位的寄存器和算术逻辑部件，16 位处理器拥有16 位的寄存器和算术逻辑部件，64 位处理器则包含64 位的寄存器和算术逻辑部件。尽管内存的最小组成单位是字节，但是，经过精心的设计和安排，它能够按字节、字、双字和四字进行访问。换句话说，仅通过单次访问就能处理8 位、16 位、32 位或者64 位的二进制数。注意，我说的是单次访问，而不是一个一个地取出每个字节，然后加以组合。

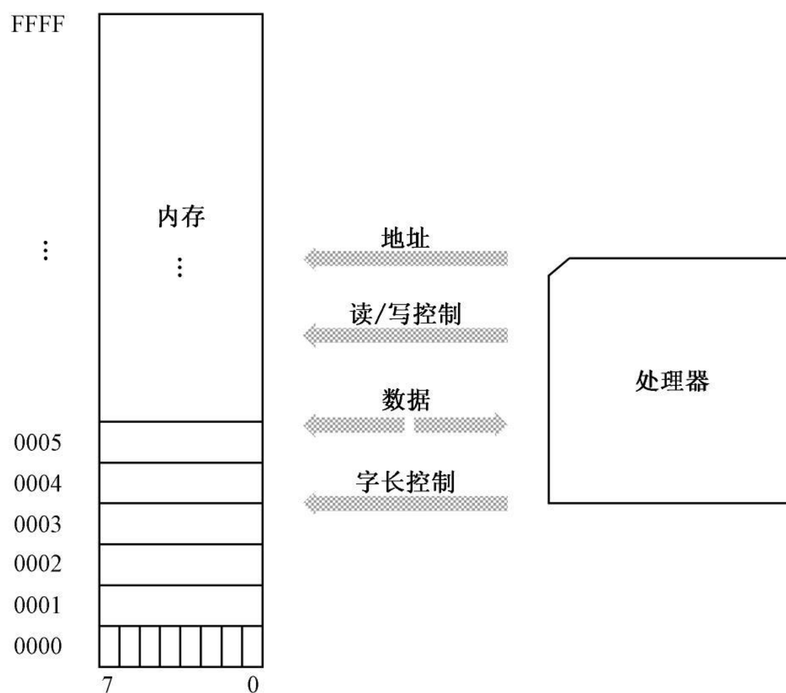


图2-5　内存和内存访问示意图

如图2-5 所示，处理器发出字长控制信号，以指示本次访问的字长是8、16、32 还是64。如果字长是8，而且给出的地址是0002H，那么，本次访问只会影响到内存的一字节；如果字长是16，给出的地址依然是0002H，那么实际访问的将是地址0002H 处的一个字，低8 位在0002H 中，高8 位在0003H 中。

检测点2.1

1．　一个字含有（ ）个字节和（ ）比特？一个双字含有（ ）个字节、（ ）个字和（ ）个比特？

2．　二进制数10000000 中，位（ ）的那个比特是“1”，也就是第（ ）位。它是最低位还是最高位？

3．一个存储器的容量是16 个字节，地址范围为（ ）~（ ）。用该存储器保存字数据时，可存放（ ）个字，这些字的地址分别是（ ），保存双字呢？

2.4　指令和指令集

从一开始，设计处理器的目标之一就是使它成为一种可以自动进行操作的器件。另外，还需要提供一种机制，来允许程序员决定进行何种操作。

处理器何以能够自动进行操作，这不是本书的话题，大学里有这样的课程，《穿越计算机的迷雾》这本书也给出了通俗化的答案。

简单地说， 处理器的设计者用某些数来指示处理器所进行的操作，这称为指令（Instruction），或者叫机器指令，因为只有处理器才认得它们。前面已经说了，处理器内部有寄存器和负责运算的部件，控制器“分析”一个个指令，然后确定在哪个时间点让哪些部件进行工作。比如，指令F4H 表示让处理器停机，当处理器取到并执行这条指令后，就停止工作。指令是集中存放在内存里的，一条接着一条，处理器的工作是自动按顺序取出并加以执行。

如图2-6 所示，从内存地址0000H 开始（也就是内存地址的最低端）连续存放了一些指令。同时，假定执行这些指令的是一个16 位处理器，拥有两个16 位的寄存器RA 和RB。

一般来说，指令由操作码和操作数构成，但也有小部分指令仅有操作码，而不含操作数。如图2-6 所示，停机指令仅包含1 字节的操作码F4，而没有操作数。指令的长度不定，短的指令仅有1 字节，而长的指令则有可能达到15 字节（对于INTEL x86 处理器来说）。

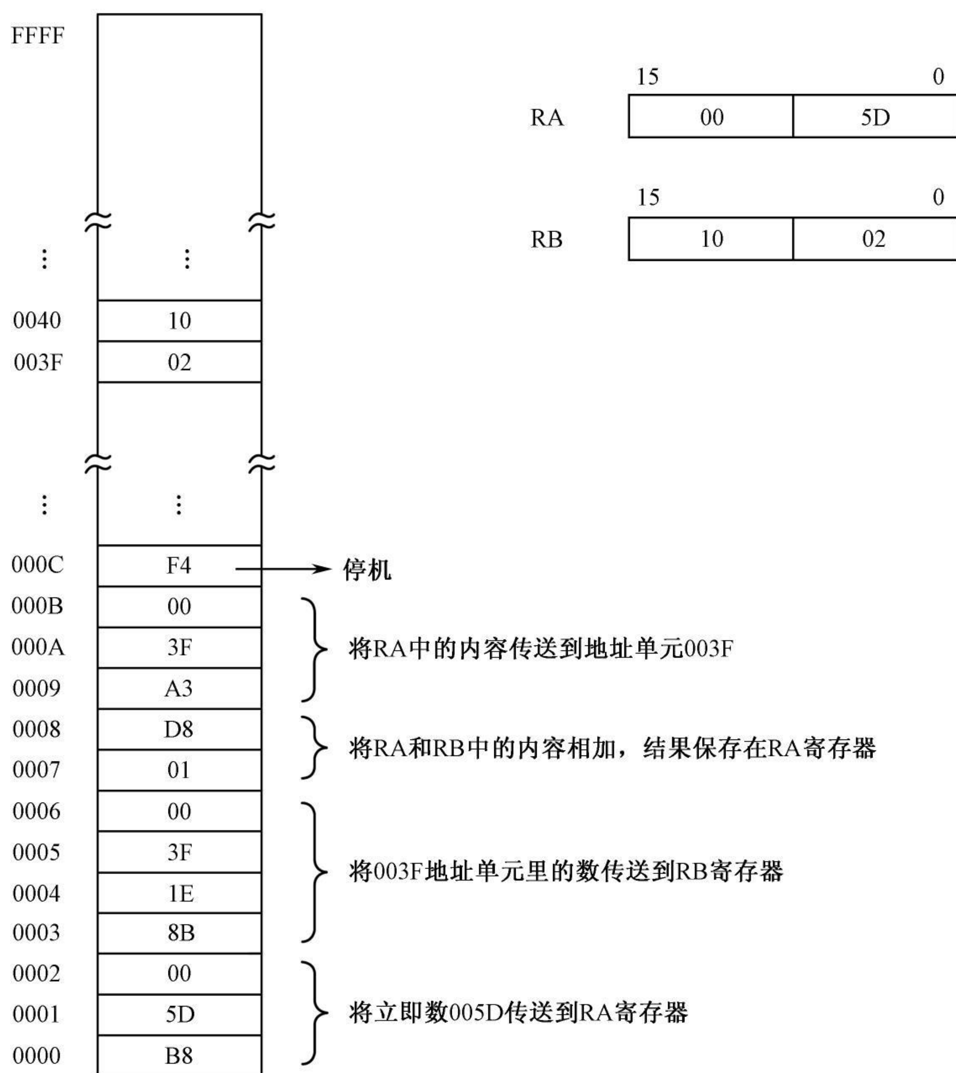


图2-6　处理器指令在内存中的布局

对处理器来说，指令的操作码隐含了如何执行该指令的信息，比如它是做什么的，以及怎么去做。第一条指令的操作码是B8，这表明，该指令是一条传送指令，第一个操作数是寄存器，第二个操作数是直接包含在指令中的，紧跟在操作码之后，可以立即从指令中取得，所以叫做立即数（Immediate Operand）。同时，操作码还直接指出该寄存器是RA。RA 是16 位寄存器，这条指令将按字进行操作。所以，当这条指令执行之后，该指令的操作数（立即数）005DH就被传送到RA 中。

既然操作码中隐含了这么多的信息，那么，处理器就可以“知道”每条指令的长度。这样，当它执行第一条指令B8 5D 00 的时候，就已经知道，这是一个3 字节指令，下一条指令位于3 个字节之后，即内存地址0003H 处。

注意字数据在内存中的存放特点。地址0001H 和0002H 里的内容分别是5D 和00，如果每次读一个字节，则从地址0001H 里读出的是5D，从0002H 里读出的是00。但如果以字的方式来访问地址0001H，读到的就会是005DH。这种差别，跟处理器和内存之间的数据线连接方式有关。对于Intel 处理器来说，如果访问内存中的一个字，那么，它规定高字节位于高地址部分，低字节位于低地址部分，这称为低端字节序（Little Endian）。至于其他公司的处理器，则可能情况正好相反，称为高端字节序。

对于复杂一些的指令来说，1 个字节的操作码可能不会够用。所以，第2 条指令的操作码为8B 1E，它隐含的意思是，这是一条传送指令，第一个操作数是寄存器，而且是RB 寄存器，第二个操作数是内存地址，要传送到RB 寄存器中的数存放在该地址中。同时，这是一个字操作指令，应当从第二个操作数指定的地址中取出一个字。

该指令的操作数部分是3F 00，指定了一个内存地址003FH。它相当于高级语言里的指针，当处理器执行这条指令时，会再次用003FH 作为地址去访问内存，从那里取出一个字（1002H），然后将它传送到寄存器RB。注意，“传送”这个词带有误导性。其实，传送的意思更像是“复制”，传送之后，003FH 单元里的数据还保持原样。

通过这两条指令的比较，很容易分清指令中的“立即数”是什么意思。指令执行和操作的对象是数。如果这个数已经在指令中给出了，不需要再次访问内存，那这个数就是立即数，比如第一条指令中的005DH；相反，如果指令中给出的是地址，真正的数还需要用这个地址访问内存才能得到，那它就不能称为立即数，比如第二条指令中的003FH。

如图2-6 所示，余下的三条指令，旁边都有注解，这里就不再一一解释了。如果一开始内存地址003FH 中存放的是1002H，那么，当所有这些指令执行完后，003FH 里就是最终的结果105FH。

指令和非指令的普通二进制数是一模一样的，在组成内存的电路中，都是一些高低电平的组合。因为处理器是自动按顺序取指令并加以执行的，在指令中混杂了非指令的数据会导致处理器不能正常工作。为此，指令和数据要分开存放，分别位于内存中的不同区域，存放指令的区域叫代码区，存放数据的区域叫数据区。为了让处理器正确识别和执行指令，工程技术人员必须精心安排，并告诉处理器要执行的指令位于内存中的什么位置。

还是那句话，并非每一个二进制数都代表着一条指令。每种处理器在设计的时候，也只能拥有有限的指令，从几十条到几百条不等。一个处理器能够识别的指令的集合，称为该处理器的指令集。

检测点2.2

在内存中，指令和数据一模一样，都是无差别的数。如图2-6 所示，假如处理器访问内存时是按低端字节序的，那么，从地址0008H 处取出一个字时，该字的值为（ ）。

2.5 古老的Intel 8086 处理器

任何时候，一旦提到Intel 公司的处理器，就不能不说8086。8086 是Intel 公司第一款16 位处理器，诞生于1978 年，所以说它很古老。

但是，在Intel 公司的所有处理器中，它占有很重要的地位，是整个Intel 32 位架构处理器（IA-32）的开山鼻祖。首先，最重要的一点是，它是一款非常成功的产品，设计先进，功能很强，卖得很好。

其次，8086 的成功使得市场上出现了大量针对它开发的软件产品。这样，当Intel 公司要设计新的处理器时，它不得不考虑到兼容性的问题。要使得老的软件也能在新的处理器上很好地运行，必须要具备指令集和工作模式上的兼容性和一致性。Intel 公司很清楚，如果新处理器和老处理器不兼容，那么，新处理器越多，它扔掉的拥趸也就越多，要不了多久，这公司就不用再开了。

所以，当我们讲述处理器的时候，必须要从8086 开始；而且，要学习汇编语言，针对8086的汇编技术也是必不可少的。

2.5.1 8086 的通用寄存器

8086 处理器内部有8 个16 位的通用寄存器，分别被命名为AX、BX、CX、DX、SI、DI、BP、SP。“通用”的意思是，它们之中的大部分都可以根据需要用于多种目的。

如图2-7 所示，因为这8 个寄存器都是16 位的，所以通常用于进行16 位的操作。比如，可以在这8 个寄存器之间互相传送数据，它们之间也可以进行算术逻辑运算；也可以在它们和内存单元之间进行16 位的数据传送或者算术逻辑运算。

同时，如图2-7 所示，这8 个寄存器中的前4 个，即AX、BX、CX 和DX，又各自可以拆分成两个8 位的寄存器来使用，总共可以提供8 个8 位的寄存器AH、AL、BH、BL、CH、CL、DH 和DL。这样一来，当需要在寄存器和寄存器之间，或者寄存器和内存单元之间进行8 位的数据传送或者算术逻辑运算时，使用它们就很方便。

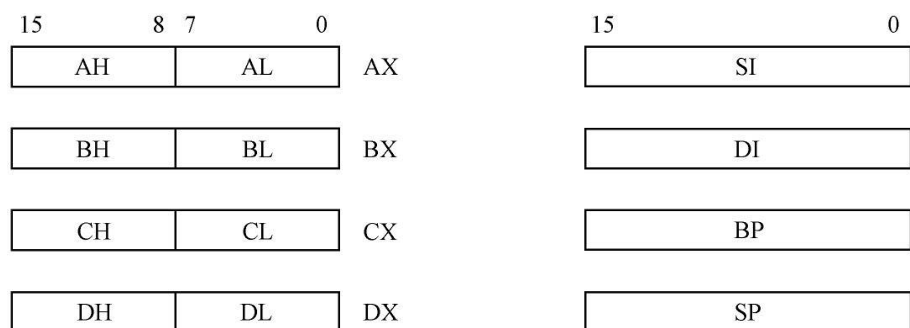


图2-7　8086 的通用寄存器

将一个16 位的寄存器当成两个8 位的寄存器来用时，对其中一个8 位寄存器的操作不会影响到另一个8 位寄存器。举个例子来说，当你操作寄存器AL 时，不会影响到AH 中的内容。

2.5.2　程序的重定位难题

我们知道，处理器是自动化的器件，在给出了起始地址之后，它将从这个地址开始，自动地取出每条指令并加以执行。只要每条指令都正确无误，它就能准确地知道下一条指令的地址。这就意味着，完成某个工作的所有指令，必须集中在一起，处于内存的某个位置，形成一个段，叫做代码段。事情是明摆着的，要是指令并没有一条挨着一条存放，中间夹杂了其他非指令的数据，处理器将因为不能识别而出错。

为了做某件事而编写的指令，它们一起形成了我们平时所说的程序。程序总要操作大量的数据，这些数据也应该集中在一起，位于内存中的某个地方，形成一个段，叫做数据段。

注意，我们并没有改变内存的物理性质，并不是真的把它分成几块。段的划分是逻辑上的，从本质上来说，是如何看待和组织内存中的数据。

段在内存中的位置并不重要，因为处理器是可控的，我们可以让它从内存的任何位置开始取指令并加以执行。这里有一个例子，如图2-8 所示，我们有一大堆数字，现在想把它们加起来求出一个总和。

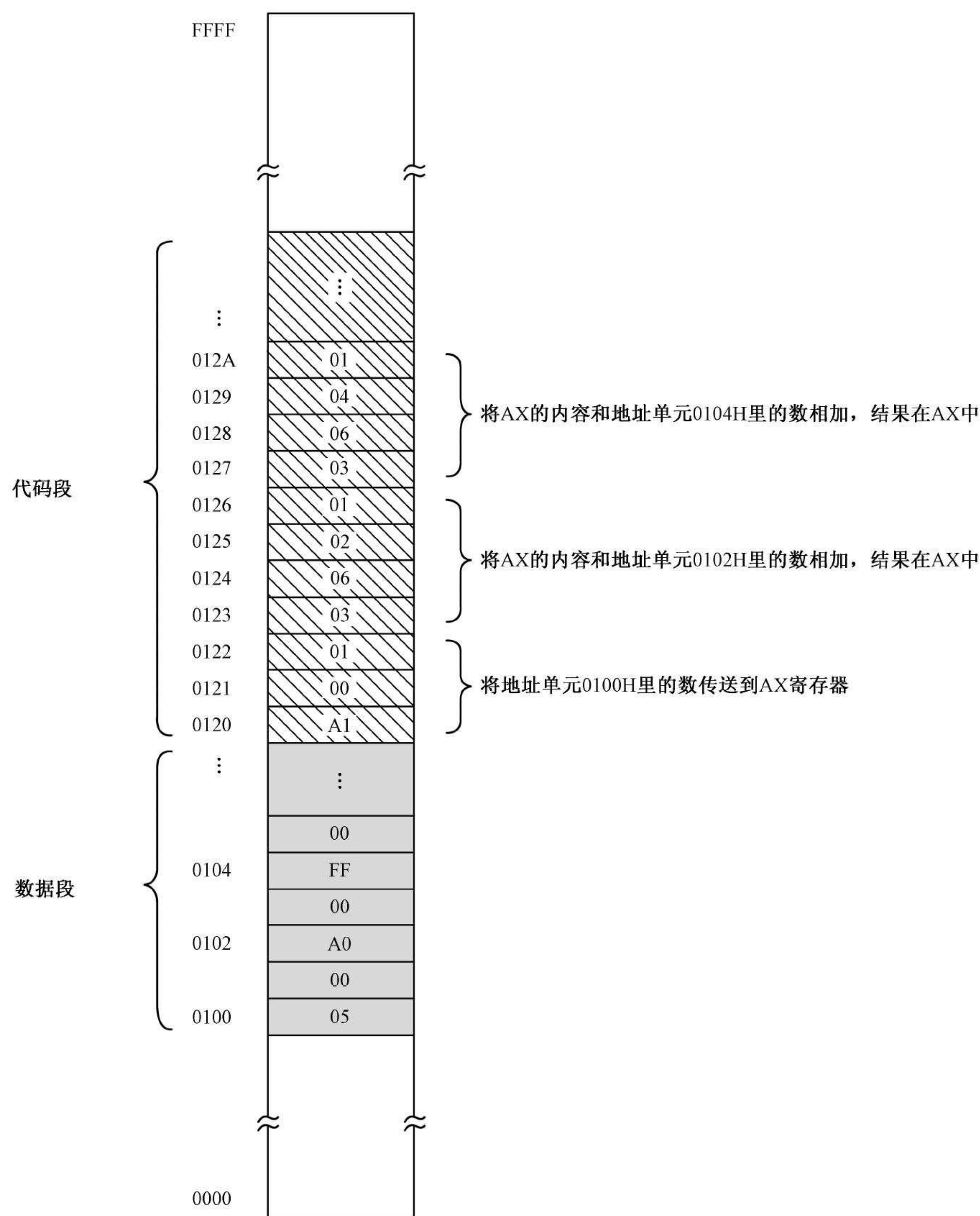


图2-8　程序的代码段和数据段示例

假定我们有16 个数要相加，这些数都是16 位的二进制数，分别是0005H、00A0H、00FFH、…。为了让处理器把它们加起来，我们应该先在内存中定义一个数据段，将这些数字写进去。数据段可以起始于内存中的任何位置，既然如此，我们将它定在0100H 处。这样一来，第一

个要加的数位于地址0100H，第二个要加的数位于地址0102H，最后一个数的地址是011EH。

一旦定义了数据段，我们就知道了每个数的内存地址。然后，紧挨着数据段，我们从内存地址0120H 处定义代码段。严格地说，数据段和代码段是不需要连续的，但这里把它们挨在一起更自然一些。为了区别数据段和代码段，我们使用了不同的底色。

代码段是从内存地址0120H 处开始的，第一条指令是A1 00 01，其功能是将内存单元0100H里的字传送到AX 寄存器。指令执行后，AX 的内容为0005H。

第二条指令是03 06 02 01，功能是将AX 中的内容和内存单元0102H 里的字相加，结果在AX 中。由于AX 的内容为0005H，而内存地址0102H 里的数是00A0H，这条指令执行后，AX 的内容为00A5H。

第三条指令是03 06 04 01，功能是将AX 中的内容和内存单元0104H 里的字相加，结果在AX 中。此时，由于AX 里的内容是00A5H，内存地址0104H 里的数是00FFH，本指令执行后，AX 的内容为01A4H。

后面的指令没有列出，但和前2 条指令相似，依次用AX 的内容和下一个内存单元里的字相加，一直到最后，在AX 中得到总的累加和。在这个例子中，我们没有考虑AX 寄存器容纳不下结果的情况。当累加的总和超出了AX 所能表示的数的范围（最大为FFFFH，即十进制的65535）时，就会产生进位，但这个进位被丢弃。

在内存中定义了数据段和代码段之后，我们就可以命令处理器从内存地址0120H 处开始执行。当所有的指令执行完后，就能在AX 寄存器中得到最后的结果。

看起来没有什么问题，一切都很完美，不是吗？那本节标题中所说的难题又从何而来呢？

这里确实有一个难题。

在前面的例子中，所有在执行时需要访问内存单元的指令，使用的都是真实的内存地址。比如A1 00 01，这条指令的意思是从地址为0100H 的内存单元里取出一个字，并传送到寄存器AX。在这里，0100H 是一个真实的内存地址，又称物理地址。

整个程序（包括代码段和数据段）在内存中的位置，是由我们自己定的。我们把数据段定在0100H，把代码段定在0120H。

问题是，大多数时候，整个程序（包括代码段和数据段）在内存中的位置并不是我们能够决定的。请想一想你平时是怎么使用计算机的，你所用的程序，包括那些用来调整计算机性能的工具、小游戏、音乐和视频播放器等，都是从网上下载的，位于你的硬盘、U 盘或光盘中。即使有些程序是你自己编写的，那又如何？当你双击它们的图标，使它们在Windows 里启动之前，内存已经被塞了很多东西，就算你是刚刚打开计算机，Windows 自己已经占用了很多内存空间，不然的话，你怎么可能在它上面操作呢？

在这种情况下，你所运行的程序，在内存中被加载的位置完全是随机的，哪里有空闲的地方，它就会被加载到哪里，并从那里开始被处理器执行。所以，前面那段程序不可能恰好如你所愿，被加载到内存地址0100H，它完全可能被加载到另一个不同的位置，比如1000H。但是，同样是那个程序，一旦它在内存中的位置发生了改变，灾难就出现了。

如图2-9 所示，因为程序现在是从内存地址1000H 处被加载的，所以，数据段的起始地址为1000H。这就是说，第一个要加的数，其地址为1000H，第二个则为1002H，其他以此类推。代码段依然紧挨着数据段之后，起始地址相应地是1020H。

只要所有的指令都是连续存放的，代码段位于内存中的什么地方都可以正常执行。所以，处理器可以按你的要求，从内存地址1020H 处连续执行，但结果完全不是你想要的。

请看第一条指令A1 00 01，它的意思是从内存地址0100 处取得一个字，将其传送到寄存器AX。但是，由于程序刚刚改变了位置，它要取的那个数，现在实际上位于1000H，它取的是别人地盘里的数！

这能怪谁呢？发生这样的事情，是因为我们在指令中使用了绝对内存地址（物理地址），这样的程序是无法重定位的。为了让你写的程序在卖给别人之后，可以在内存中的任何地方正确执行，就只能在编写程序的时候使用相对地址或者逻辑地址了，而不能使用真实的物理地址。当程序加载时，这些相对地址还要根据程序实际被加载的位置重新计算。

在任何时候，程序的重定位都是非常棘手的事情。当然，也有好几种解决的办法。在8086 处理器上，这个问题特别容易解决，因为该处理器在访问内存时使用了分段机制，我们可以借助该机制。

2.5.3 内存分段机制

如图2-10 所示，整个内存空间就像长长的纸条，在内存中分段，就像从长纸条中裁下一小段来。根据需要，段可以开始于内存中的任何位置，比如图中的内存地址A532H 处。

在这个例子中，分段开始于地址为A532H的内存单元处，这个起始地址就是段地址。

这个分段包含了6个存储单元。在分段之前，它们在整个内存空间里的物理地址分别是A532H 、 A533H 、 A534H 、 A535H 、 A536H、A537H。

但是，在分段之后，它们的地址可以只相对于自己所在的段。这样，它们相对于段开始处的距离分别为0、1、2、3、4、5，这叫做偏移地址。

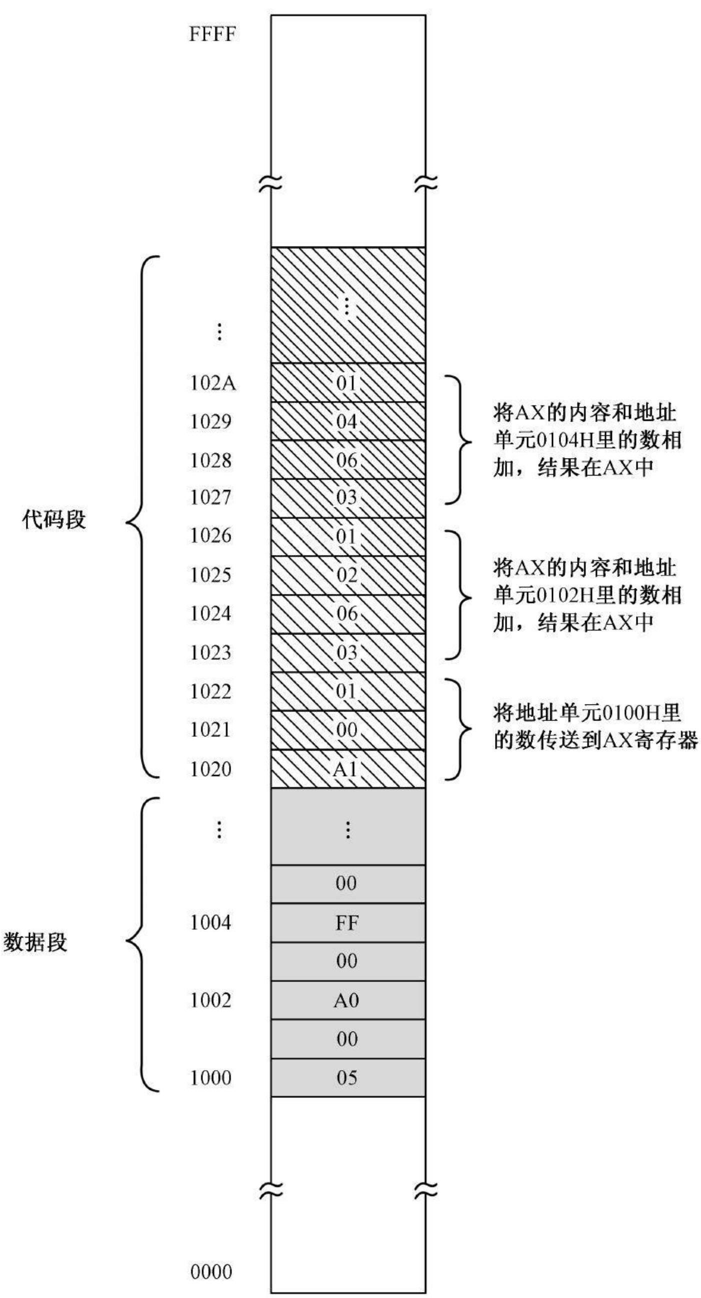


图2-9 在指令中使用绝对内存地址的程序是不可重定位的

于是，当采用分段策略之后，一个内存单元的地址实际上就可以用“段：偏移”或者 “段地址：偏移地址”来表示，这就是通常所说的逻辑地址。比如，在图2-10 中，段内第1 个存储单元的地址为A532H:0000H，第3 个存储单元的地址为A532H:0002H，而本段最后一个存储单元的地址则是A532H:0005H。

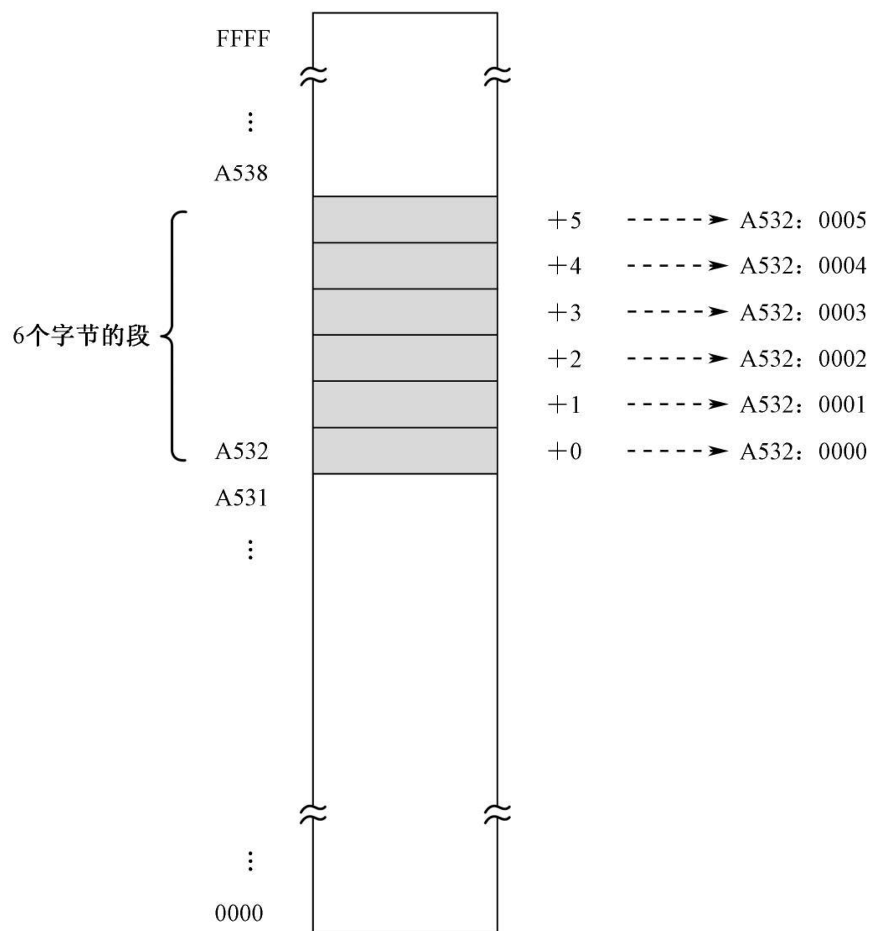


图2-10　段地址和偏移地址示意图

为了在硬件一级提供对“段地址：偏移地址”内存访问模式的支持，处理器至少要提供两个段寄存器，分别是代码段寄存器（Code Segment，CS）和数据段寄存器（Data Segment，DS）。

对CS 内容的改变将导致处理器从新的代码段开始执行。同样，在开始访问内存中的数据之前，也必须首先设置好DS 寄存器，使之指向数据段。

除此之外，最重要的是，当处理器访问内存时，它把指令中指定的内存地址看成是段内的偏移地址，而不是物理地址。这样，一旦处理器遇到一条访问内存的指令，它将把DS 中的数据段起始地址和指令中提供的段内偏移相加，来得到访问内存所需要的物理地址。

如图2-11 所示，代码段的段地址为1020H，数据段的段地址为1000H。在代码段中有一条指令A1 02 00，它的功能是将地址0002H 处的一个字传送到寄存器AX。在这里，处理器将0002H 看成是段内的偏移

地址，段地址在DS 中，应该在执行这条指令之前就已经用别的指令传送到DS 中了。

当执行指令A1 02 00 时，处理器将把DS 中的内容和指令中指定的偏移地址0002H 相加，得到1002H。这是一个物理地址，处理器用它来访问内存，就可以得到所需要的数00A0H。

如果一下次执行这个程序时，代码段和数据段在内存中的位置发生了变化，只要把它们的段地址分别传送到CS 和DS，它也能够正确执行。

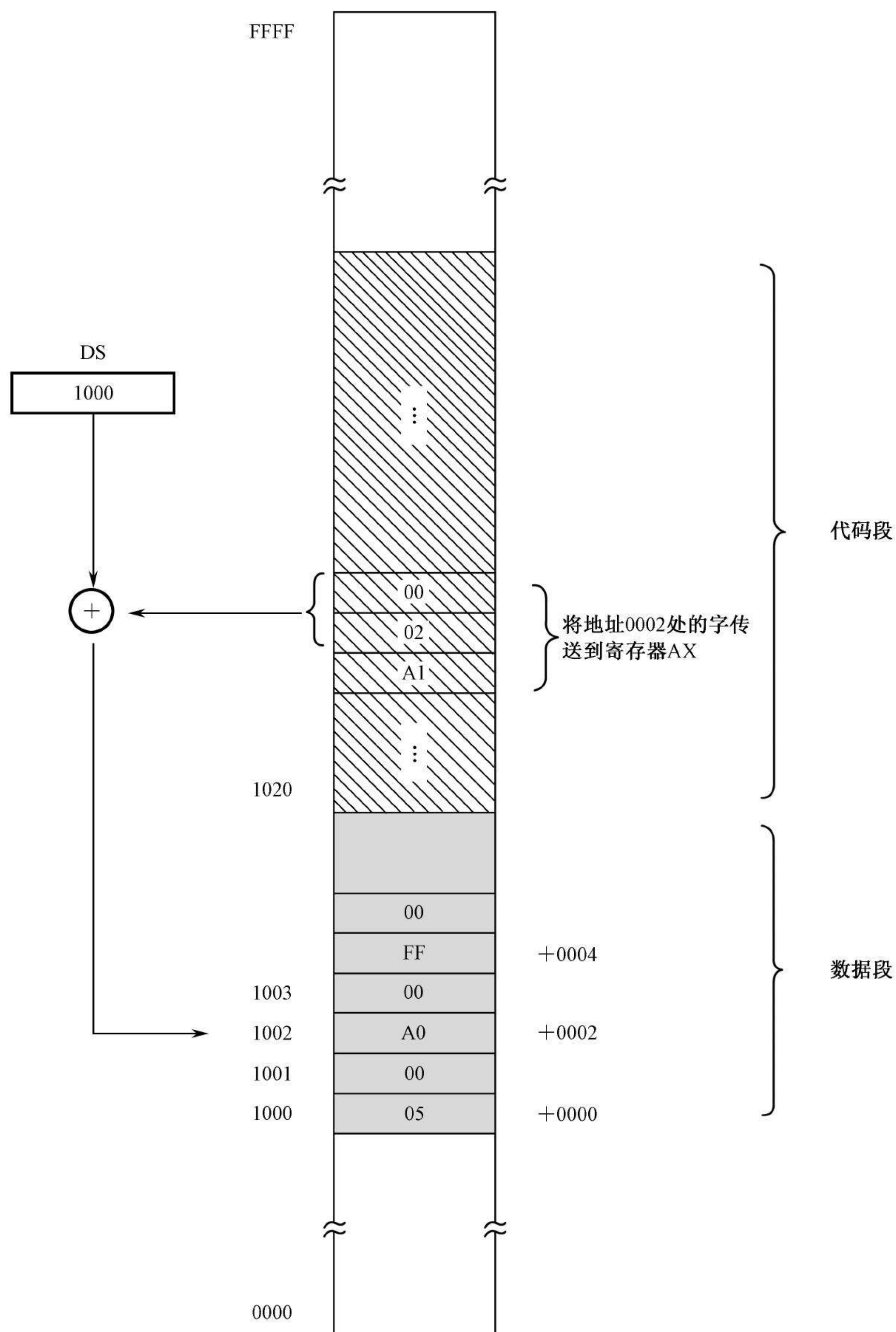


图2-11　从逻辑地址到物理地址的转换过程

2.5.4　8086的内存分段机制

前面讲了如何从逻辑地址转换到物理地址，以使得程序的运行和它在内存中的位置无关。这种策略在很多处理器中得到了支持，包括8086处理器。但是，由于8086 自身的局限性，它的做法还要复杂一些。

如图2-12 所示，8086 内部有8 个16 位的通用寄存器，分别是AX、BX、CX、DX、SI、DI、BP、SP。其中，前四个寄存器中的每一个，都还可以当成两个8 位的寄存器来使用，分别是AH、AL、BH、BL、CH、CL、DH、DL。

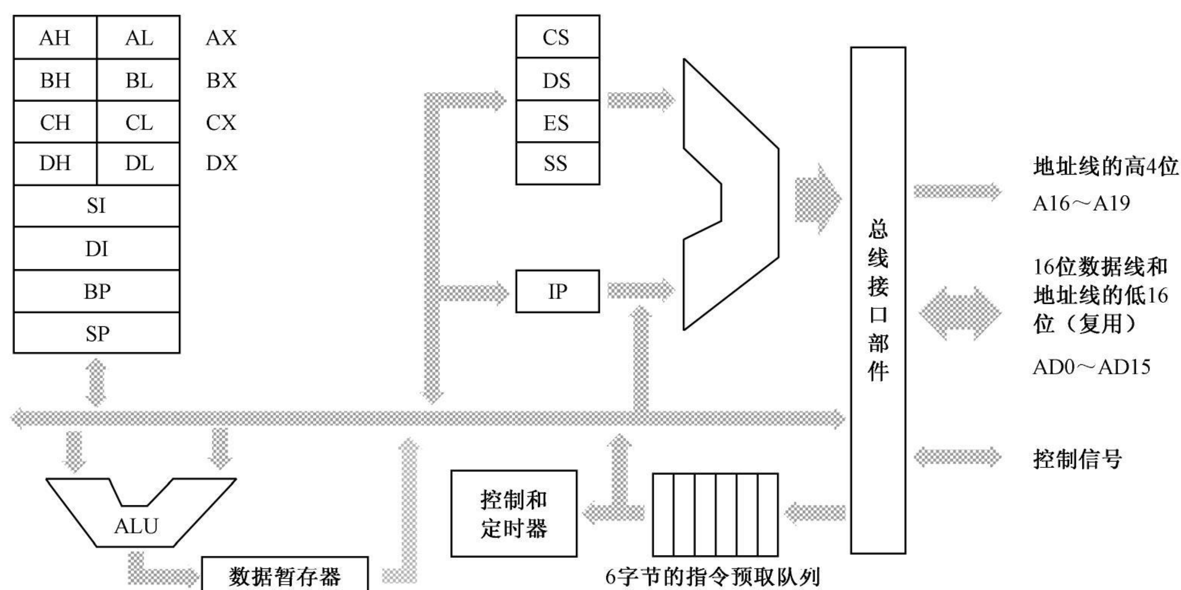


图2-12　8086 处理器内部组成框图

在进行数据传送或者算术逻辑运算的时候，使用算术逻辑部件（ALU）。比如，将AX 的内容和CX 的内容相加，结果仍在AX 中，那么，在相加的结果返回到AX 之前，需要通过一个叫数据暂存器的寄存器中转。

处理器能够自动运行，这是控制器的功劳。为了加快指令执行速度，8086 内部有一个6 字节的指令预取队列，在处理器忙着执行那些不需要访问内存的指令时，指令预取部件可以趁机访问内存预取指令。这时，多达6 个字节的指令流可以排队等待解码和执行。

8086 内部有4 个段寄存器。其中，CS 是代码段寄存器，DS 是数据段寄存器，ES 是附加段（Extra Segment）寄存器。附加段的意思是，它是额外赠送的礼物，当需要在程序中同时使用两个数据段时，DS 指向一个，ES 指向另一个。可以在指令中指定使用DS 和ES 中的哪一个，

如果没有指定，则默认是使用DS。SS 是栈段寄存器，以后会讲到，而且非常重要。

IP 是指令指针（Instruction Pointer）寄存器，它只和CS 一起使用，而且只有处理器才能直接改变它的内容。当一段代码开始执行时，CS 指向代码段的起始地址，IP 则指向段内偏移。这样，由CS 和IP 共同形成逻辑地址，并由总线接口部件变换成物理地址来取得指令。然后，处理器会自动根据当前指令的长度来改变IP 的值，使它指向下一条指令。

当然，如果在指令的执行过程中需要访问内存单元，那么，处理器将用DS 的值和指令中提供的偏移地址相加，来形成访问内存所需的物理地址。

8086 的段寄存器和IP 寄存器都是16 位的，如果按照原先的方式，把段寄存器的内容和偏移地址直接相加来形成物理地址的话，也只能得到16 位的物理地址。麻烦的是，8086 却提供了20根地址线。换句话说，它提供的是20 位的物理地址。

提供20 位地址线的原因很简单，16 位的物理地址只能访问64KB 的内存，地址范围是0000H～FFFFH，共65536 个字节。这样的容量，即使是在那个年代，也显得捉襟见肘。注意，这里提到了一个表示内存容量的单位“KB”。为了方便，我们通常使用更大的单位来描述内存容量，比如千字节（KB）、兆字节（MB）和吉字节（GB），它们之间的换算关系如下：

```
1 KB = 1024 Byte
1 MB = 1024 KB
1 GB = 1024 MB
```

所以，65536 个字节就是64KB，而20 位的物理地址则可以访问多达1MB 的内存，地址范围从00000H 到FFFFFH。问题是，16 位的段地址和16 位的偏移地址相加，只能形成16 位的物理地址，怎么得到这20位的物理地址呢？

为了解决这个问题，8086 处理器在形成物理地址时，先将段寄存器的内容左移4 位（相当于乘以十六进制的10，或者十进制的16），形成20 位的段地址，然后再同16 位的偏移地址相加，得到20 位的物理地址。比如，对于逻辑地址F000H:052DH，处理器在形成物理地址时，将

段地址F000H 左移4 位，变成F0000H，再加上偏移地址052DH，就形成了20 位的物理地址F052DH。

这样，因为段寄存器是16 位的，在段不重叠的情况下，最多可以将1MB 的内存分成65536个段，段地址分别是0000H、0001H、0002H、0003H，……，一直到FFFFH。在这种情况下，如图2-13 所示，每个段正好16 个字节，偏移地址从0000H 到000FH。

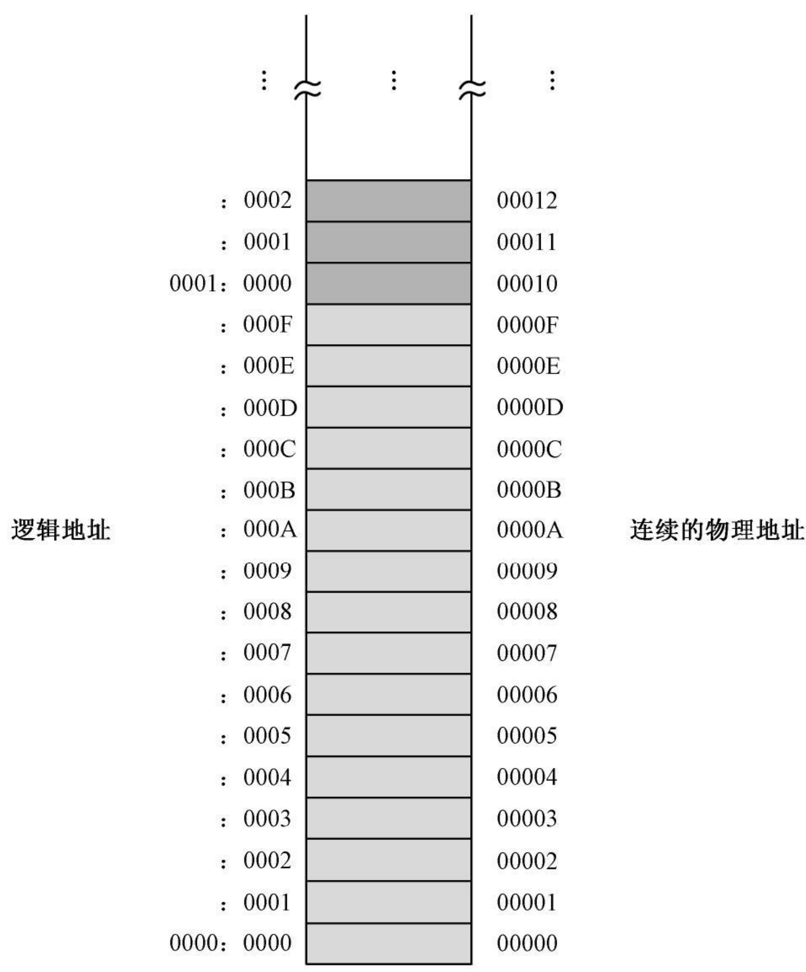


图2-13　1MB 内存可以划分为65536 个16 字节的段

同样在不允许段之间重叠的情况下，每个段的最大长度是64KB，因为偏移地址也是16 位的，从0000H 到FFFFH。在这种情况下，1MB 的内存，最多只能划分成16 个段，每段长64KB，段地址分别是0000H、1000H、2000H、3000H，…，一直到F000H。

以上所说的只是两种最典型的情况。通常情况下，段地址的选择取决于内存中哪些区域是空闲的。举个例子来说，假如从物理地址00000H

开始，一直到82251H 处都被其他程序占用着，而后面一直到FFFFFH 的地址空间都是自由的，那么，你可以从物理内存地址82251H 之后的地方加载你的程序。

接着，你的任务是定义段地址并设置处理器的段寄存器，其中最重要的是段地址的选取。因为偏移地址总是要求从0000H 开始，而82260H 是第一个符合该条件的物理地址，因为它恰好对应着逻辑地址8226H:0000H，符合偏移地址的条件，所以完全可以将段地址定为8226H。

但是，举个例子来说，如果你从物理内存地址82255H 处加载程序，由于它根本无法表示成一个偏移地址为0000H 的逻辑地址，所以不符合要求，段不能从这里开始划分。这里面的区别在于，82260H 可以被十进制数16（或者十六进制数10H）整除，而82255H 不能。通过这个例子可以看出，8086 处理器的逻辑分段，起始地址都是16 的倍数，这称为是按16 字节对齐的。

段的划分是自由的，它可以起始于任何16 字节对齐的位置，也可以是任意长度，只要不超过64KB。比如，段地址可以是82260H，段的长度可以是64KB。在这种情况下，该段所对应的逻辑地址范围是8226H:0000H～8226H:FFFFH，其所对应的物理地址范围是82260～9225FH。

同时，正是由于段的划分非常自由，使得8086 的内存访问也非常随意。同一个物理地址，或者同一片内存区域，根据需要，可以随意指定一个段来访问它，前提是那个物理地址位于该段的64KB 范围内。也就是说，同一个物理地址，实际上对应着多个逻辑地址。

检测点2.3

1． INTEL 8086 处理器有（ ）个16 位通用寄存器，分别是（ ）。其中，有些还可以分开来作为两个独立的8 位寄存器来用，这几个8 位寄存器分别是（ ）。

2． 选择题（可多选）：INTEL 8086 处理器取指令时，使用段寄存器（ ）和指令指针寄存器（ ）。方法是，将段寄存器的值（ ），加上指令指针寄存器的当前值，形成物理地址访问内存。

A.CS　B.DS　C.IP　D.左移4位　E.右移4位　F.乘以1　G.除以10H

3． 物理地址132FEH 对应的逻辑地址是（可多选）：

A.132FH:000EH B.1300H:02FEH C.1000H:32FEH D.1320H:00FEH

E.102FH:03E0H F.0FE0H:34FEH

本章习题

1．在段与段之间互不重叠的前提下，1MB 内存可以完整地划分为多少个16KB 的段？

2．数据段寄存器DS 的值为25BCH 时，计算Intel 8086可以访问的物理地址范围。

第3章　汇编语言和汇编软件

处理器依靠机器指令工作，但机器指令从形式上看都是一些没有规律的数字，难以书写、阅读和理解，这样就发明了汇编语言。本章的目标是：

1．了解汇编语言的作用和“汇编”一词的由来。

2．下载NASM 编译器，并学会使用它来编译汇编语言源程序。

3.1 汇编语言简介

在前面的章节里，我们讲到了处理器，也讲了处理器是如何进行算术逻辑运算的。为了实现自动计算，处理器必须从内存中取得指令，并执行这些指令。

指令和被指令引用的数据在内存中都是一些或高或低的电平，每一个电平都可以看成是一个二进制位（0 或者1），8 个二进制位形成一字节。

要解读内存中的东西，最好的办法就是将它们按字节转换成数字的形式。比如，下面这些数字就是存放在内存中的INTEL8086 指令，我们用的是十六进制：

```
B8  3F  00  01  C3  01  C1
```

对于大多数人来说，他们很难想象上面那一排数字对应着下面几条 8086 指令：

```
将立即数 003FH 传送到寄存器 AX；
将寄存器 BX 的内容和寄存器 AX 的内容相加，结果在 BX 中；
将寄存器 CX 的内容和寄存器 AX 的内容相加，结果在 CX 中。
```

即使是很有经验的技术人员，要想用这种方式来编写指令，也是很困难的，而且很容易出错。所以，在第一个处理器诞生之后不久，如何使指令的编写变得更容易，就提上了日程。

为了克服机器指令难以书写和理解的缺点，人们想到可以用一些容易理解和记忆的符号，也就是助记符，来描述指令的功能和操作数的类型，这就产生了汇编语言（Assembly Language）。这样，上面那些指令就可以写成：

```
mov ax,3FH
add bx,ax
add cx,ax
```

对于那些有点英语基础的人来说，理解这些汇编语言指令并不困难。比如这句

```
mov ax,3FH
```

首先，mov 是move 的简化形式，意思是“移动”或者“传送”。至于“ax”，很明显，指的就是AX 寄存器。传送指令需要两个操作数，分别是目的操作数和源操作数，它们之间要用逗号隔开。在这里，AX 是目的操作数，源操作数是3FH。汇编语言对指令的大小写没有特别的要求。所以你完全可以这样写：

```
MOV AX,3FH
```

```
mov ax,3fh
MOV ax,3FH
mov AX,3fh
```

在很多高级语言中，如果要指示一个数是十六进制数，通常不采用在后面加“H”的做法，而是为它添加一个“0x”前缀。像这样：

```
mov ax,0x3f
```

你可能想问一下，为什么会是这样，为什么会是“0x”？答案是不知道，不知道在什么时候，为什么就这样用了。这不得不让人怀疑，它肯定是一个非常随意的决定，并在以后形成了惯例。如果你知道确切的答案，不妨写封电子邮件告诉我。注意，为了方便，我们将在本书中采用这种形式。

在汇编语言中，使用十进制数是最自然的。因为3FH 等于十进制数63，所以你可以直接这样写：

```
mov ax,63
```

当然，如果你喜欢，也可以使用二进制数来这样写：

```
mov ax,00111111B
```

一定要看清楚，在那串“0”和“1”的组合后面，跟着字母“B”，以表明它是一个二进制数。

至于这句：

```
    add bx,ax
```

情况也是一样。add 的意思是把一个数和另一个数相加。在这里，是把BX 寄存器的内容和AX 寄存器的内容相加。相加的结果在BX 中，但AX 的内容并不改变。

像上面那样，用汇编语言提供的符号书写的文本，叫做汇编语言源程序。为此，你需要一个字处理器软件，比如Windows 记事本，来编辑这些内容。如图3-1 所示，相信这些软件的使用都是你已经非常熟悉的。



图3-1　用Windows 记事本来书写汇编语言源程序

有了汇编语言所提供的符号，这只是方便了你自己。相反地，对人类来说通俗易懂的东西，处理器是无法识别的。所以，还需要将汇编语言源程序转换成机器指令，这个过程叫做编译（Compile）。在编译的时候，汇编语言编译器的作用是将mov、add、ax、bx 等这些符号组合起来，转换成类似于数值的机器指令，这个过程叫做汇编，这就是汇编语言的由来，也有人称之为组合语言。

编译肯定还需要依靠一个软件，称为编译器，或编译软件。因为如果需要人类自己去做，还费这周折干嘛。另一方面，想想看，一个帮助人类生产软件的工具，自己居然也是一个软件，这很有意思。

从字处理器软件生成的是汇编语言源程序文件。编译软件的任务是读取这些文件，将那些符号转变成二进制形式的机器指令代码。它把这

些机器代码存放到另一个文件中，叫做二进制文件或者可执行文件，比如Windows 里以“.exe”为扩展名的文件，就是可执行文件。当需要用处理器执行的时候，再加载到内存里。

3.2 NASM 编译器

3.2.1 NASM 的下载和安装

每种处理器都可能会有自己的汇编语言编译器，而对于同一款处理器来说，针对不同的平台（比如Windows 和Linux），也会有不同版本的汇编语言编译器。

现存的汇编语言编译器有多种，用得比较多的有MASM、FASM、TASM、AS86、GASM等，每种汇编器都有自己的特色和局限性。特别是，有些还需要付费才能使用。不同于前面所列举的这些，在本书中，我们用的是另一款叫做NASM 的汇编语言编译器。

NASM 的全称是Netwide Assembler，它是可免费使用的开源软件。下面是它的下载地址：

```
http://sourceforge.net/projects/nasm/files/
```

通过以上地址可以找到所有平台上的NASM 版本，比如为16 位和32位DOS、LINUX、OS/2等操作系统开发的版本。因为本书的读者一般在Windows 平台上工作，所以应当使用下面的链接来直接定位到Windows平台上的NASM 版本：

```
http://sourceforge.net/projects/nasm/files/Win32%20binaries/
```

通过以上链接，可以显示所有Windows 平台上的NASM 版本，应当选择最新版本下载。这本书出版的时候，最新的NASM 版本是2.07。

本书不配光盘，所以书中的所有源代码连同我自己写的小工具都只有通过网络下载方能使用。这是一个压缩文件，名字叫booktool.zip，可以通过下面这个链接来下载它：

```
http://ishare.iask.sina.com.cn/f/34697012.html
```

如果此链接不可用，可以到电子工业出版社的网站上下载，或者直接给我写信，我会以最快的时间给予支持。

3.2.2 代码的书写和编译过程

和你已经司空见惯的其他Windows 应用程序不同，NASM 在运行之后并不会显示一个图形用户界面。相反地，它只能通过命令行使用。

比如，我们可以用Windows 记事本编写一个汇编语言源程序，并把它保存到NASM 工作目录下（就是在前面安装NASM 时所用的安装文件夹），文件名为exam.asm。作为惯例，汇编语言源程序文件的扩展名是“.asm”，不过，你当然可以使用其他扩展名。

一旦有了一个源程序，下一步就是将它的内容编译成机器代码。为此，可以从Windows 开始菜单里找到“Netwide Assembler xxx”，其中的“xxx”取决于你安装的NASM 版本。然后，选择其下的“Nasm Shell”，这将打开一个命令行窗口。

接着，在命令行提示符后输入“nasm –f bin exam.asm –o exam.bin”并按Enter 键，如图3-2所示。

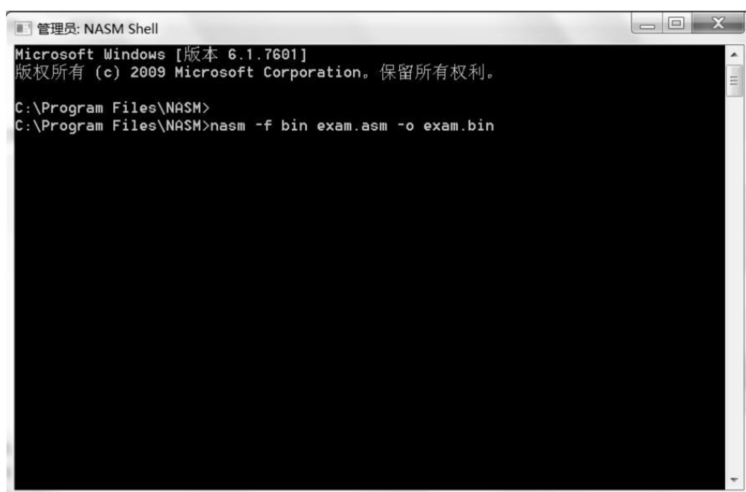


图3-2　在命令行方式下使用NASM 编译一个汇编源程序

NASM 需要一系列参数才能正常工作。-f 参数的作用是指定输出文件的格式（Format）。这样，-f bin 就是要求NASM 生成的文件只包含“纯二进制”的内容。换句话说，除了处理器能够识别的机器代码外，别的任何东西都不包含。这样一来，因为缺少操作系统所需要的加载和重定位信息，它就很难在Windows、DOS 和Linux 上作为一个普通的应用程序运行。不过，这正是本书所需要的。

紧接着，exam.asm 是源程序的文件名，它是将要被编译的对象。

-o 参数指定编译后输出（Output）的文件名。在这里，我们要求NASM 生成输出文件exam.bin。

用来编写汇编语言源程序，Windows 记事本并不是一个好工具。同时，在命令行编译源程序也令很多人迷糊。毕竟，很多年轻的朋友都是用着Windows 成长起来的，他们缺少在DOS 和UNIX 下工作的经历。

为了写这本书，我一直想找一个自己中意的汇编语言编辑软件。互联网是个大宝库，上面有很多这样的工具软件，但大多都包含了太多的功能，用起来自然也很复杂。我的愿望很简单，能够方便地书写汇编指令即可，同时还具有编译功能。毕竟我自己也不喜欢在命令行和图形用户界面之间来回切换。

在经历了一系列的失望之后，我决定自己写一个，于是就有了Nasmide 这个小程序，它同样位于配书文件包中。不过遗憾的是，这个小程序却并非是用汇编语言书写的。

现在，你可以双击nasmide.exe 来运行它。启动之后，如图3-3 所示，Nasmide 的软件界面分为三个部分。顶端是菜单，可以用来新建文件、打开文件、保存文件或者调用NASM 来编译当前文档。



图3-3　Nasmide 程序的基本界面

中间最大的空白区域是编辑区，用来书写汇编语言源代码。

窗口底部那个窄的区域是消息显示区。在编译当前文档时，不管是编译成功，还是发现了文档中的错误，都会显示在这里。

基本上，你现在已经可以在Nasmide 里书写汇编语句了。不过，在此之前你最好先做一件事情。Nasmide 只是一个文本编辑工具，它自己

没有编译能力。不过，它可以在后台调用NASM 来编译当前文档，前提是它必须知道NASM 安装在什么地方。

为此，你需要在菜单上选择“选项”→“编译环境设置”来打开如图3-4所示的对话框。

如图3-4 所示，这个路径就是你在前面安装NASM 时，指定的安装路径，包括可执行文件名nasm.exe。

不同于其他汇编语言编译器，NASM 最让我喜欢的一个特点是允许在源程序中只包含指令，如图3-5 所示。用过微软公司MASM 的人都知道，在真正开始书写汇编指令前，先要穿靴戴帽，在源程序中定义很多东西，比如代码段和数据段等，弄了半天，实际上连一条指令还没开始写呢。

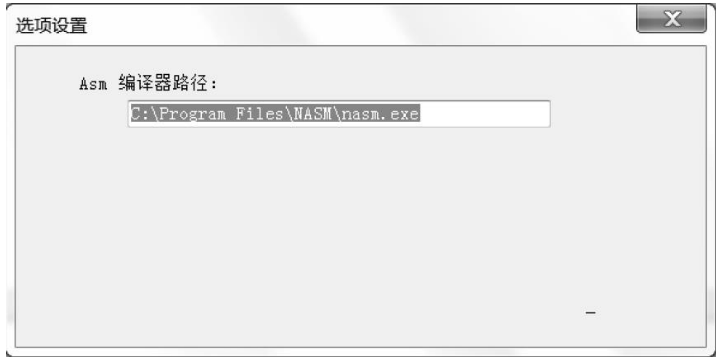


图3-4　为Nasmide 指定NASM 编译器所在路径

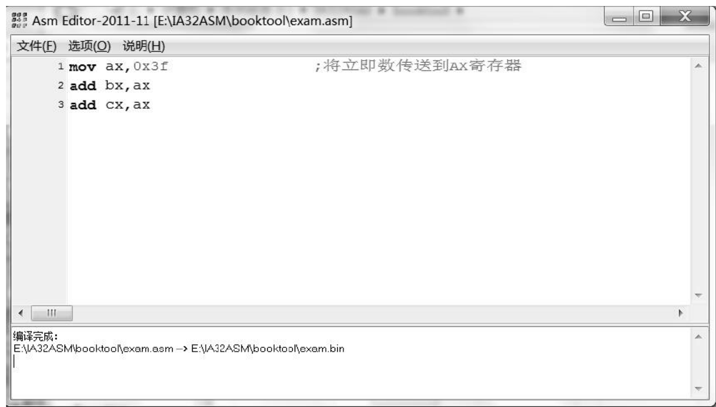


图3-5　NASM 允许在源文件中只包含指令

如图3-5 所示，用Nasmide 程序编辑源程序时，它会自动在每一行内容的左边显示行号。对于初学者来说，一开始可能会误以为行号也会

出现在源程序中。不要误会，行号并非源程序的一部分，当保存源程序的时候，也不会出现在文件内容中。

让Nasmide 显示行号，这是一个聪明的决定。一方面，我在书中讲解源程序时，可以说第几行到第几行是做什么用的；另一方面，当编译源程序的时候，如果发现了错误，错误信息中也会说明是第几行有错。这样，因为Nasmide 显示了行号，这就很容易快速找到出错的那一行。

在汇编源程序中，可以为每一行添加注释。注释的作用是说明某条指令或者某个符号的含义和作用。注释也是源程序的组成部分，但在编译的时候会被编译器忽略。如图3-5 所示，为了告诉编译器注释是从哪里开始的，注释需要以英文字母的分号“;”开始。

当源程序书写完毕之后，就可以进行编译了，方法是在Nasmide 中选择菜单“文件”→“编译本文档”。这时，Nasmide 将会在后台调用NASM来完成整个编译过程，不需要你额外操心。如图3-5所示，即使只有三行的程序也能通过编译。编译完成后，会在窗口底部显示一条消息。

检测点3.1

1． 在你的计算机中启动Nasmide 程序，输入图3-10 中的三行代码，然后编译它们。看看消息显示区是否有编译成功的提示。

2． 选择填空：指令mov ax,0xf5fc 中，“mov”指示这是一条（ ）指令，0xf5fc 是（ ）。指令执行后，寄存器AX 中的内容是（ ）。

A.立即数 B.传送 C.0xf5fc D.加法 E.0xfcf5 F.寄存器

3.2.3 用HexView 观察编译后的机器代码

编译成功完成之后，Nasmide 会在编辑窗口的底部显示相应的消息，同时显示了源文件名称和编译之后的文件名称（含路径）。

尽管我们强调源文件和编译之后的文件具有不同的内容，但如果能用工具看一看，相信印象更为深刻。在前面下载的配书源码和工具里，有一个名为HexView 的小程序，可以实现这个愿望。HexView 用于打开任意一个文件，以十六进制的形式从头到尾显示它每个字节的内容。

双击启动HexView，然后选择菜单“文件”→“打开文件以显示”，在文件选择对话框里找到你在3.2.4 节里编辑并保存的源程序文件。

如图3-6 所示，文件选择之后，HexView 程序将以十六进制的形式显示刚刚选择的文件。

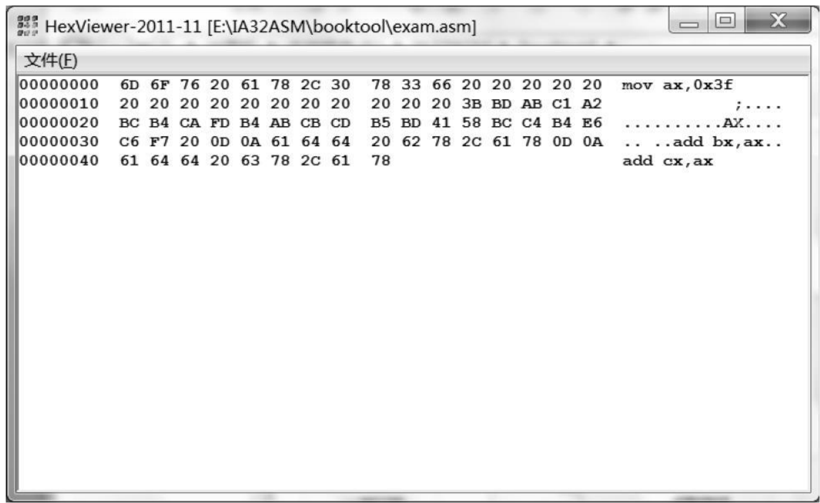


图3-6　用HexView 程序显示源程序文件的内容

在HexView 中，文件的内容以十六进制的形式显示在窗口中间，以16 个字节为一行，字节之间以空白分隔，所以看起来很稀疏。如果文件较大的话，则会分成很多行。

作为对照，每个字节还会以字符的形式显示在窗口右侧，如果它确实可显示为一个字符的话。如果该字节并非一个可以显示的字符，则显示一个替代的字符“.”。因为源程序中还有汉字注释，所以，如果细心的话，从图中可以算出每个汉字的编码是两个字节，比如“将”字的编码是0xBD 0xAB。由于HexView 以单字节的形式来显示每个字符，所以无法显示汉字。

左边的数字，是每一行第一个字节相对于文件头部的距离（偏移），也是以十六进制数显示的。字母“m”是整个源程序文件内的第1 个字符，因此，它的偏移量是00000000（H），其他字符以此类推，最后一个字符“x”的偏移量是00000048（H）。

源程序很长，但是，编译之后的机器指令却很简短。

如图3-7 所示，编译之后的文件只有7 个字节，这才是处理器可以识别并执行的机器指令。

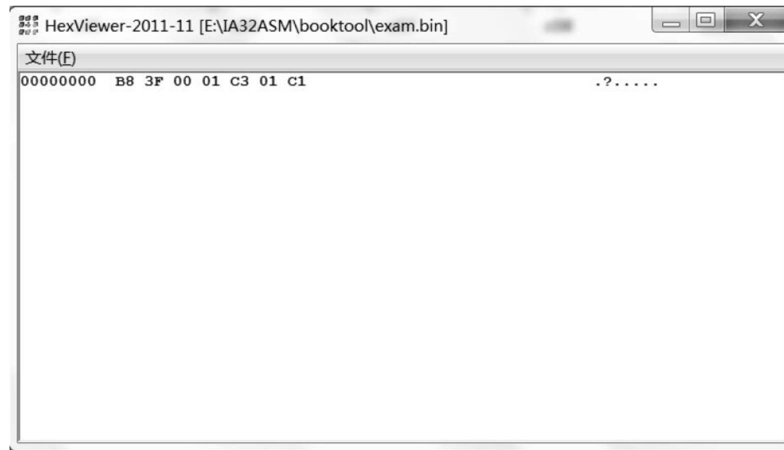


图3-7　用HexView 显示编译之后的文件内容

本章习题

如图3-6 所示，请问：

（1）源程序共有3 行，每一行第一个字符在文件内的偏移量分别是多少？

（2）该源程序文件的大小是多少字节？

第4章　虚拟机的安装和使用

和其他所有计算机语言一样，汇编语言程序设计具有很强的实践性，不实际上机操作，不思考，不能举一反三，是无法掌握它的。但是，当程序编译完成后，如何让处理器执行它呢？还有，如何才能知道执行的结果是不是正确呢？这都是非常重要的问题，要在本章里解决。本章的目标是：

1．了解计算机的开机启动过程。只有这样，你才能知道我们应当把编译好的程序放到哪里才会被处理器执行到。

2．了解硬盘的构造和作用。

3．了解VirtualBox 虚拟机的功能，下载和安装VirtualBox 虚拟机软件，创建一台本书中要用到的虚拟机，学会往虚拟硬盘中写数据，学会VirtualBox 虚拟机的使用方法。

4.1　计算机的启动过程

4.1.1　如何将编译好的程序提交给处理器

对于绝大多数编译好的程序来说，要想得到处理器的光顾，让它执行一下，必须借助于操作系统。就拿Windows 来说，它为你显示每个程序的图标，允许你双击来运行它们。在内部你看不见的层面上，它必须给将要运行的程序分配空闲的内存空间，并在适当的时候将程序提交给处理器执行。

每种操作系统都对它所管理的程序提出了种种格式上的要求。比如，它要求编译好的程序必须在文件的开始部分包含编译日期，是针对哪种操作系统编译的，程序的版本，第一条指令从哪里开始，数据段从哪里开始、有多长，代码段从哪里开始、有多长，等等，Windows 甚至建议你在文件中包含至少一个用于显示的图标。如果你不按它的要求来，它也不会给你面子，并直截了当地弹出一个对话框，如图4-1 所示，告诉你它不准备，也没办法将你的程序提交给处理器。

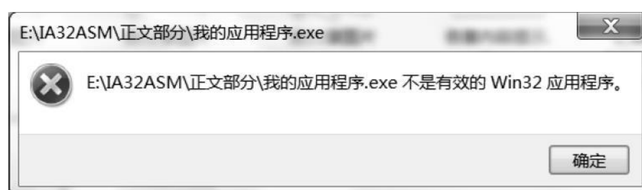


图4-1　每种操作系统都会定义它自己的可执行文件格式

每种编译器都有能力针对不同的操作系统来生成不同格式的二进制文件，程序员所要做的，就是在源程序中加入一些相关的信息，比如指定每个段的开始和结束，并在编译时指定适当的参数。如果你对此感兴趣，可以阅读NASM 文档。这是一个PDF 文件，在安装NASM 的时候，它也会被安装。

在特定的操作系统上开发软件肯定不是一件容易的事情。但换个角度考虑一下，操作系统也是一个需要在处理器上运行的软件，只不过比起一般的程序而言，体积更为庞大，功能更为复杂而已。如果我们能绕过它，或者代替它，让计算机一开机的时候直接执行我们自己的软件，岂不更简单？

好，这个主意完全可行。那就让我们慢慢开始吧。

4.1.2　计算机的加电和复位

在处理器众多的引脚中，有一个是RESET，用于接受复位信号。每当处理器加电，或者RESET 引脚的电平由低变高时[1]，处理器都会执行一个硬件初始化，以及一个可选的内部自测试（Build-in Self-Test，BIST），然后将内部所有寄存器的内容初始到一个预置的状态。

比如，对于Intel 8086 来说，复位将使代码段寄存器（CS）的内容为0xFFFF，其他所有寄存器的内容都为0x0000，包括指令指针寄存器（IP）。8086 之后的处理器并未延续这种设计，但毫无疑问，无论怎么设计，都是有目的的。

处理器的主要功能是取指令和执行指令，加电或者复位之后，它就会立刻尝试去做这样的工作。不过，在这个时候，内存中还没有任何有意义的指令和数据，它该怎么办呢？

在揭开谜底之前，我们先来看看内存的特点。

为了节约成本，并提高容量和集成度，在内存中，每个比特的存储都是靠一个极其微小的晶体管，外加一个同样极其微小的电容来完成的。可以想象，这样微小的电容，其泄漏电荷的速度当然也非常快。所以，个人计算机中使用的内存需要定期补充电荷，这称为刷新，所以这种存储器也称为动态随机访问存储器（Dynamic Random Access Memory，DRAM）。随机访问的意思是，访问任何一个内存单元的速度和它的位置（地址）无关。举个例子来说，从头至尾在一盘录音带上找某首歌曲，它越靠前，找到它所花的时间就越短。但内存就不一样，读写地址为0x00001 的内存单元，和读写地址为0xFFFF0 的内存单元，所需要的时间是一样的。

在内存刷新期间，处理器将无法访问它。这还不是最麻烦的，最麻烦的是，在它断电之后，所有保存的内容都会统统消失。所以，每当处理器加电之后，它无法从内存中取得任何指令。

4.1.3　基本输入输出系统

Intel 8086 可以访问1MB 的内存空间，地址范围为0x00000 到0xFFFFF。出于各方面的考虑，计算机系统的设计者将这1MB 的内存空间从物理上分为几个部分。

8086 有20 根地址线，但并非全都用来访问DRAM，也就是内存条。事实上，这些地址线经过分配，大部分用于访问DRAM，剩余的部分给了只读存储器ROM 和外围的板卡，如图4-2 所示。

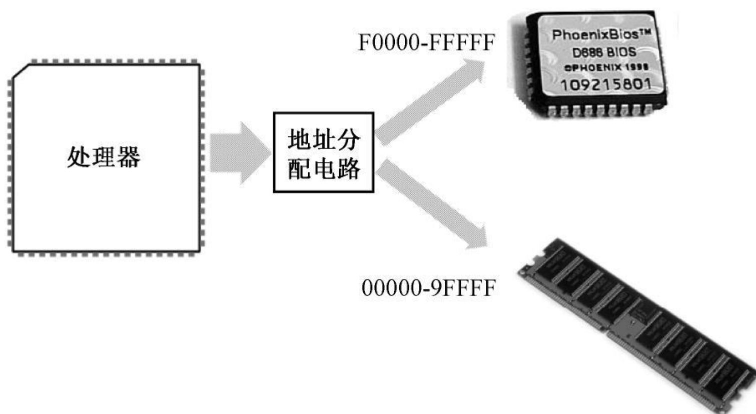


图4-2　8086 系统的内存空间分配

与DRAM 不同，只读存储器（Read Only Memory，ROM）不需要刷新，它的内容是预先写入的，即使掉电也不会消失，但也很难改变。这个特点很有用，比如，可以将一些程序指令固化在ROM 中，使处理器在每次加电时都自动执行。处理器醒来后不能饿着，这是很重要的。

在以Intel 8086 为处理器的系统中，ROM 占据着整个内存空间顶端的64KB，物理地址范围是0xF0000～0xFFFFF，里面固化了开机时要执行的指令；DRAM 占据着较低端的640KB，地址范围是0x00000～0x9FFFF；中间还有一部分，分给了其他外围设备，这个以后再说。因为8086 加电或者复位时，CS=0xFFFF，IP=0x0000，所以，它取的第一条指令位于物理地址0xFFFF0，正好位于ROM 中，那里固化了开机时需要执行的指令。

处理器取指令执行的自然顺序是从内存的低地址往高低地址推进。如果从0xFFFF0 开始执行，这个位置离1MB 内存的顶端（物理地址0xFFFFF）只有16 个字节的长度，一旦IP 寄存器的值超过0x000F，比如IP=0x0011，那么，它与CS 一起形成的物理地址将因为溢出而变成0x00001，这将回绕到1MB 内存的最低端。

所以，ROM 中位于物理地址0xFFFF0 的地方，通常是一个跳转指令，它通过改变CS 和IP的内容，使处理器从ROM 中的较低地址处开始取指令执行。在NASM 汇编语言里，一个典型的跳转指令像这样：

```
jmp 0xf000:0xe05b
```

在这里，“jmp”是跳转（jump）的简化形式；0xf000 是要跳转到的段地址，用来改变CS 寄存器的内容；0xe05b 是目标代码段内的偏移地址，用来改变IP 寄存器的内容。因此，目标位置的物理地址是0xfe05b。一旦执行这条指令，处理器将开始从指定的“段: 偏移”处开始重新取指令执行。

到了本书第5 章我们就能接触跳转指令了，现在，我们只需要知道，指令的执行并非总是顺序的，有时候不得不根据某些条件来选择执行哪些指令，不执行哪些指令。这个时候，跳转指令是很有用的。

这块ROM 芯片中的内容包括很多部分，主要是进行硬件的诊断、检测和初始化。所谓初始化，就是让硬件处于一个正常的、默认的工作状态。最后，它还负责提供一套软件例程，让人们在不必了解硬件细节的情况下从外围设备（比如键盘）获取输入数据，或者向外围设备（比如显示器）输出数据。设备当然是很多的，所以这块ROM 芯片只针对那些最基本的、对于使用计算机而言最重要的设备，而它所提供的软件例程，也只包含最基本、最常规的功能。正因为如此，这块芯片又叫基本输入输出系统（Base Input & Output System，BIOS）ROM。在读者缺乏基础知识的情况下讲述ROM-BIOS 的工作只会越讲越糊涂，所以这些知识将会分散在各个章节里予以讲解。

ROM-BIOS 的容量是有限的，当它完成自己的使命后，最后所要做的，就是从辅助存储设备读取指令数据，然后转到那里开始执行。基本上，这相当于接力赛中的交接棒。

4.1.4 硬盘及其工作原理

历史上，有多种辅助存储设备，比如软盘、光盘、硬盘、U 盘等，相对于内存，它们就是人们常说的“外存”，即外存储器（设备）。

从软盘（Floppy Disk）启动计算机，这已经是过去的事了。软盘的尺寸比烟盒稍大一点，但是比较薄，采用塑料作为基片，上面是一层磁

性物质，可以用来记录二进制位。这种塑料介质比较柔软，所以称为软盘。

在数据记录原理上和软盘很相似的设备是硬盘（Hard Disk，HDD），而且它们几乎是同一个时代的产物。但是，与软盘不同，硬盘是多盘片、密封、高转速的，采用铝合金作为基片，并在表面涂上磁性物质来记录二进制位。这就使得它的盘片具有较高的硬度，故称为硬盘。

如图4-3 所示，这是一块被拆开的硬盘，中间是用于记录数据的铝合金盘片，固定在中心的轴上，由一个高速旋转的马达驱动。附着在盘片表面的扁平锥状物，就是用于在盘片上读写数据的磁头。



图4-3　一块被拆开密封盖的硬盘

为了进一步搞清楚硬盘的内部构造，图4-4 给出了更为详细的图示。

硬盘可以只有一个盘片（这称为单碟），也可能有好几个盘片。但无论如何，它们都串在同一个轴上，由电动机带动着一起高速旋转。一般来说，转速可以达到每分钟3600 转或者7200转，有的能达到一万多转，这个参数就是我们常说的"转/分钟"（Round Per Minute，RPM）。

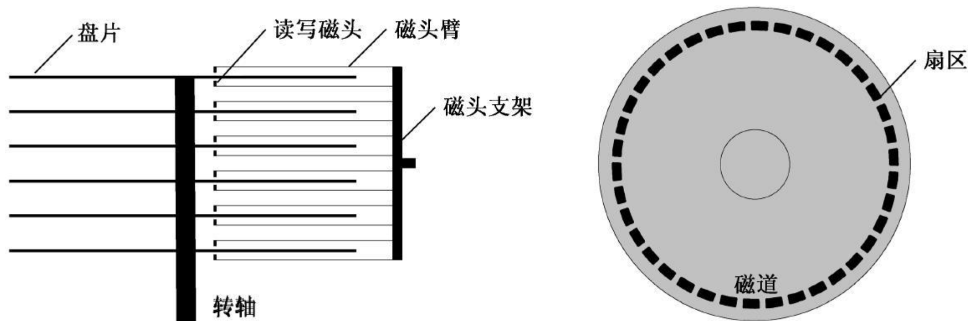


图4-4　硬盘的结构示意图

每个盘片都有两个磁头（Head），上面一个，下面一个，所以经常用磁头来指代盘面。磁头都有编号，第1 个盘片，上面的磁头编号为0，下面的磁头编号为1；第2 个盘片，上面的磁头编号为2，下面的磁头编号为3，以此类推。

每个磁头不是单独移动的。相反，它们都通过磁头臂固定在同一个支架上，由步进电动机带动着一起在盘片的中心和边缘之间来回移动。也就是说，它们是同进退的。步进电动机由脉冲驱动，每次可以旋转一个固定的角度，即可以步进一次。

可以想象，当盘片高速旋转时，磁头每步进一次，都会从它所在的位置开始，绕着圆心“画”出一个看不见的圆圈，这就是磁道（Track）。磁道是数据记录的轨迹。因为所有磁头都是联动的，故每个盘面上的同一条磁道又可以形成一个虚拟的圆柱，称为柱面（Cylinder）。

磁道，或者柱面，也要编号。编号是从盘面最边缘的那条磁道开始，向着圆心的方向，从0开始编号。

柱面是一个用来优化数据读写的概念。初看起来，用硬盘来记录数据时，应该先将一个盘面填满后，再填写另一个盘面。实际上，移动磁头是一个机械动作，看似很快，但对处理器来说，却很漫长，这就是寻道时间。为了加速数据在硬盘上的读写，最好的办法就是尽量不移动磁头。这样，当0 面的磁道不足以容纳要写入的数据时，应当把剩余的部分写在1 面的同一磁道上。如果还写不下，那就继续把剩余的部分写在2 面的同一磁道上。换句话说，在硬盘上，数据的访问是以柱面来组织的。

实际上，磁道还不是硬盘数据读写的最小单位，磁道还要进一步划分为扇区（Sector）。磁道很窄，也看不见，但在想象中，它仍呈带状，占有一定的宽度。将它划分许多分段之后，每一部分都呈扇形，这就是扇区的由来。

每条磁道能够划分为几个扇区，取决于磁盘的制造者，但通常为63个。而且，每个扇区都有一个编号，与磁头和磁道不同，扇区的编号是从1 开始的。

扇区与扇区之间以间隙（空白）间隔开来，每个扇区以扇区头开始，然后是512 个字节的数据区。扇区头包含了每个扇区自己的信息，主要有本扇区的磁道号、磁头号和扇区号，用来供硬盘定位机构使用。现代的硬盘还会在扇区头部包括一个指示扇区是否健康的标志，以及用来替换该扇区的扇区地址。用于替换扇区的，是一些保留和隐藏的磁道。

4.1.5　一切从主引导扇区开始

尽管我们使用硬盘的历史很长，但它一直没能退出舞台，这主要是因为它总能通过不断提高自己的容量来打败那些竞争者。20 世纪90 年代初，40MB 的硬盘算是常见的，能拥有200MB 的硬盘很让人羡慕。看看现在，500GB 的硬盘也不算稀罕，而且价钱也很便宜。

前面说到，当ROM-BIOS 完成自己的使命之前，最后要做的一件事是从外存储设备读取更多的指令来交给处理器执行。现实的情况是，绝大多数时候，对于ROM-BIOS 来说，硬盘都是首选的外存储设备。

硬盘的第一个扇区是0 面0 道1 扇区，或者说是0 头0 柱1 扇区，这个扇区称为主引导扇区。如果计算机的设置是从硬盘启动，那么，ROM-BIOS 将读取硬盘主引导扇区的内容，将它加载到内存地址0x0000:0x7c00 处（也就是物理地址0x07C00），然后用一个jmp 指令跳到那里接着执行：

```
jmp 0x0000:0x7c00
```

为什么偏偏是0x7c00 这个地方？还不太清楚。反正当初定下这个方案的家伙已经被人说了很多坏话，我也就不准备再多说什么了。

通常，主引导扇区的功能是继续从硬盘的其他部分读取更多的内容加以执行。像Windows 这样的操作系统，就是采用这种接力的方法一步一步把自己运行起来的。

说到这里，我们可以想象，如果我们把自己编译好的程序写到主引导扇区，不也能够让处理器执行吗？

对于这种想法，我有一个好消息和一个坏消息要告诉你。

好消息是，这是可以的，而且这几乎是在不依赖操作系统的情况下，让我们的程序可以执行的唯一方法。

不过，坏消息是，如果你改写了硬盘的主引导扇区，那么，Windows 和Linux，以及任何你正在使用的操作系统都会瘫痪，无法启动了。

那么，我们该怎么办呢？答案是在你现有的计算机上，再虚拟出一台计算机来。

检测点4.1

1． 硬盘的磁头（盘面）是从数字（ ）开始编号的；每个盘面磁道是从数字（ ）开始编号的；每磁道/柱面上的扇区是从数字（ ）开始编号的，主引导扇区的位置是（ ）面（ ）道（ ）扇区；

2． 如果希望处理器从当前位置转移到物理地址0xc5030 处开始执行，可以使用下面的哪些指令（可多选）：

A. jmp 0xc000:0x5030　B. jmp 0xc500:0x0030

C. jmp 0xc503:0x0000　D. jmp 0xbb00:0xa030

4.2 创建和使用虚拟机

4.2.1 别害怕，虚拟机是软件

对于第一次听说虚拟机（Virtual Machine，VM）的人来说，可能以为还要再花钱买一台计算机，这恐怕是他们最担心的。所谓虚拟机，就是在你的计算机上再虚拟出另一台计算机来。这台虚拟出来的计算机，和真正的计算机一样，可以启动，可以关闭，还可以安装操作系统、安装和运行各种各样的软件，或者访问网络。总之，你在真实的计算机上能做什么，在它里面一样可以那么做。使用虚拟机，你会发现，在Windows 操作系统里，居然又可以拥有另一套Windows。然而本质上，它只是运行在物理计算机上的一个软件程序。

如图4-5 所示，整个大的背景，是Windows 7 的桌面，它安装在一台真实的计算机上。图中的小窗口，正是虚拟机，运行的是Windows Server 2003。像这样，我们就得到了两台“计算机”，而且它们都可以操作。

虚拟机仅仅是一个软件，运行在各种主流的操作系统上。它以自己运行的真实计算机为模板，虚拟出另一套处理器、内存和外围设备来。它的处理能力，完全来自于背后那台真实的计算机。

尤其重要的是，针对某种真实处理器所写的任何指令代码，通常都可以正确无误地在该处理器的虚拟机上执行。实际上，这也是虚拟机具有广泛应用价值的原因所在。

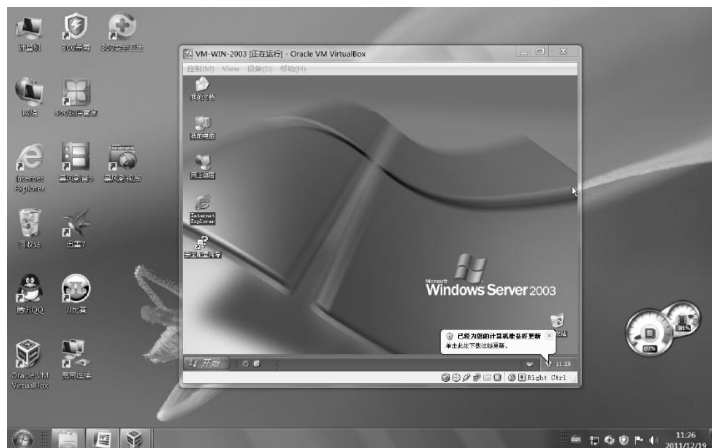


图4-5　虚拟机的实例

在过去的若干年里，虚拟机得到了广泛应用。为了研制防病毒软件、测试最新的操作系统或者软件产品，软件公司通常需要多台用于做实验的计算机。采用虚拟机，就可以避免反复重装软件系统的麻烦，当这些软件系统崩溃时，崩溃的只是虚拟机，而真实的物理计算机丝毫不受影响。

利用虚拟机来教学，本书不是第一个，国内外都流行这种教学方式。虚拟机利用软件来模拟完整的计算机系统，无须添加任何新的设备，而且与主计算机系统是隔离的，在虚拟机上的任何操作都不会影响到物理计算机上的操作系统和软件，这对拥有大量计算机的培训机构来说，可以极大地节省维护上的成本。

4.2.2　下载和安装Oracle VM VirtualBox

主流的虚拟机软件包括VMWare、Virtual PC 和VirtualBox 等，但只有VirtualBox 是开源和免费的。

要使用VirtualBox，首先必须从网上下载并安装它。这里是它的主页：

```
https://www.virtualbox.org/
```

通过这个主页，你可以找到最新的版本并下载它。为了方便，下面给出下载页面的链接：

```
https://www.virtualbox.org/wiki/Downloads
```

本书的配书文件包中提供了关于如何下载、安装和配置VirtualBox 软件的文档，有WORD 和PDF 两种版本，请选择使用。注意，要选择最新的版本下载，而且，由于该软件针对不同的操作系统平台开发，因此，要下载适用于Windows 的安装程序。

VirtualBox 软件安装完毕之后，你需要创建，或者说“虚拟”出一台计算机来，并设置该“计算机”的相关参数，包括为它配备一块硬盘。有关的方法和步骤在配书文件包的教程中已有介绍，唯一的建议是选用本书为你准备的虚拟硬盘。

和真实的计算机一样，虚拟机也需要一个或几个辅助存储器（磁盘、光盘、U 盘等）才能工作。不过，为它配备的并非真正的盘片，而是一个特殊的文件，故称为虚拟盘。这样，当一个软件程序在虚拟机里读写硬盘或者光盘时，虚拟机将把它转换成对文件的操作，而软件程序还以为自己真的是在读写物理盘片。这样的一块磁盘，在需要的时候随时创建，不需要时可以随时删除，这真是非常神奇的磁盘。

前面你已经从网上下载了与本书配套的源码和工具，那是个压缩文件。解压之后，在源代码和工具文件夹里有一个现成的虚拟硬盘文件，文件名是LEECHUNG.VHD，这是给你额外准备的，而且经过了测试，可以在你无法创建虚拟硬盘的时候派上用场。不管是你自己创建虚拟硬盘，还是选用这个现成的，都应当使虚拟硬盘文件位于源代码所在的文件夹，将来往该虚拟硬盘写数据时比较方便。

正如前面所说的，市面上有好几种流行的虚拟机软件，而每种虚拟机软件都企图制定自己的虚拟硬盘标准。因为虚拟硬盘实际是一个文件，所以，所谓虚拟硬盘标准，实际上就是该文件的格式。正是因为这样，虚拟硬盘类型说白了就是你准备采用哪家的虚拟硬盘文件格式。

因为虚拟硬盘实际上是一个文件，所以，通常来说，它的格式体现在它的文件扩展名上。比如上面的LEECHUNG.VHD，采用的就是微软公司的VHD 虚拟硬盘规范。VHD 规范最早起源于Connectix 公司的虚拟机软件Connectix Virtual PC，2003 年，微软公司收购了它并改名为Microsoft Virtual PC。2006 年，微软公司正式发布了VHD 虚拟硬盘格式规范。在本书配套的源代码和工具包里，有该规范的文档。

VDI 是VirtualBox 自己的虚拟硬盘规范，VMDK 是VMWare 的虚拟硬盘规范。采用哪个公司、哪个虚拟机软件的虚拟硬盘格式，对于普通的应用来说，这没什么关系，它们都能很好地工作。但是，对于本书和本书配套的工具来说，你必须选择“VHD（Virtual Hard Disk）”。具体原因，我们将在下一节讲述。

事实上，即使是VHD，也分为两种类型：固定尺寸的和动态分配的。一个固定尺寸的VHD，它对应的文件尺寸和该虚拟硬盘的容量是相同的，或者说是一次性分配够了的。比如，一个2GB 的VHD 虚拟硬盘，它对应的文件大小也是2GB。注意，本书以及本书配套的工具仅支持固定尺寸的VHD。

一旦完成了全部的准备工作，刚刚创建的虚拟机就会显示在VirtualBox 控制台里，如图4-6所示，虚拟机的名字叫“LEARN-ASM”。基本上，你现在就可以单击控制台界面上的“开始”来启动这台虚拟机。但是，别忙，你的虚拟硬盘里还没有东西呢。



图4-6 通过向导程序创建的LEARN-ASM 虚拟机

4.2.3 虚拟硬盘简介

坦白地说，之所以要采用固定尺寸的VHD 虚拟硬盘，是因为其简单性。我们知道，虚拟硬盘实际上是一个文件。固定尺寸的VHD 虚拟硬盘是一个具有“.vhd”扩展名的文件，它仅包括两个部分，前面是数据区，用来模拟实际的硬盘空间，后面跟着一个512 字节的结尾（2004 年前的规范里只有511 字节）。

要访问硬盘，运行中的程序必须至少向硬盘控制器提供4 个参数，分别是磁头号、磁道号、扇区号，以及访问意图（是读还是写）。

硬盘的读写是以扇区为最小单位的。所以，无论什么时候，要从硬盘读数据，或者向硬盘写数据，至少得是1 个扇区。

你可能想，我只有2 字节的数据，不足以填满一个扇区，怎么办呢？

这是你自己的事。你可以用无意义的废数字来填充，凑够一个扇区的长度，然后写入。读取的时候也是这样，你需要自己跟踪和把握从扇区里读到的数据，哪些是你真正想要的。换句话说，硬盘只是机械和电

子的组合，它不会关心你都写了些什么。要是手机像人类一样智能，它一定会在坏人使用它的时候无法开机。

在VHD 规范里，每个扇区是512 字节。VHD 文件一开始的512 字节，就对应着物理硬盘的0 面0 道1 扇区。然后，VHD 文件的第二个512 字节，对应着0 面0 道2 扇区，后面的以此类推，一直对应到0 面0 道n 扇区。这里，n 等于每磁道的扇区数。

再往后，因为硬盘的访问是按柱面进行的，所以，在VHD 文件中，紧接着前面的数据块，下一个数据块对应的是1 面0 道1 扇区，就这样一直往后排列，当把第一个柱面全部对应完后，再从第二个柱面开始对应。

如图4-7 所示，为了标志一个文件是VHD 格式的虚拟硬盘，并为使用它的虚拟机提供该硬盘的参数，在VHD 文件的结尾，包含了512 字节的格式信息。为了观察这些信息，我们使用了前面已经介绍过的配书工具HexView。

如图4-7 所示，文件尾信息是以一个字符串“conectix”开始的。这个标志用来告诉试图打开它的虚拟机，这的确是一个合法的VHD 文件。该标志称为VHD 创建者标识，就是说，该公司（conectix）创建了VHD 文件格式的最初标准。

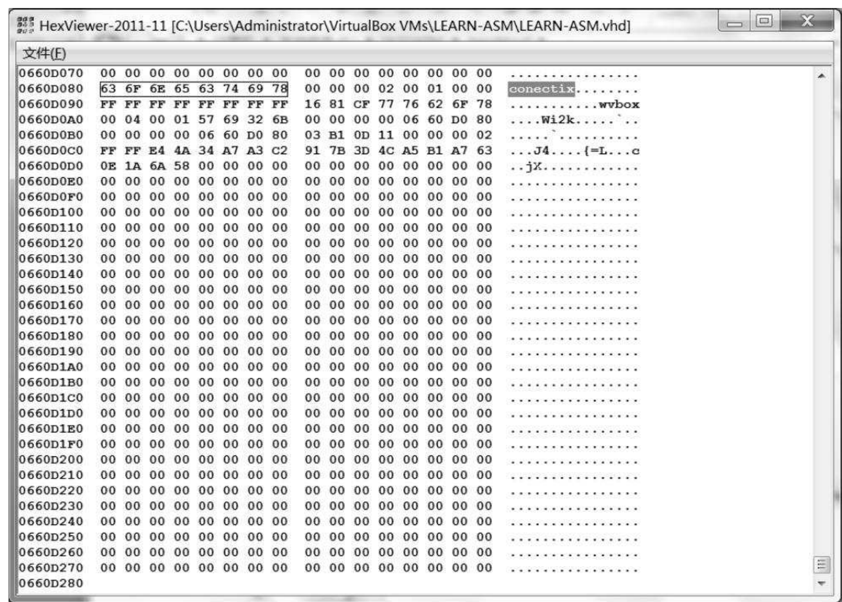


图4-7　VHD 文件的格式信息

从这个标志开始，后面的数据包含了诸如文件的创建日期、VHD 的版本、创建该文件的应用程序名称和版本、创建该文件的应用程序所属的操作系统、该虚拟硬盘的参数（磁头数、每面磁道数、每磁道扇区数）、VHD 类型（固定尺寸还是动态增长）、虚拟硬盘容量等。

说到这里，也许你已经明白我为什么要在书中使用固定尺寸的VHD。是的，因为它简单。为了学习汇编语言，我们不得不在硬盘上直接写入程序。因为VHD 格式简单，所以我只花了很少的时间就开发了一个虚拟硬盘写入程序，作为配书工具让大家使用，这就是下一节将要介绍的FixVhdWr。

至于为什么要使用VirtualBox 虚拟机，是因为它支持VHD，而且是免费的。先前版本的VirtualBox 可以识别VHD，但不支持创建新的VHD，尽管微软公司很早就公开了VHD 规范。好消息是现在的VirtualBox 也可以创建VHD 了。

4.2.4　练习使用FixVhdWr 工具向虚拟硬盘写数据

通常，VHD 是由虚拟机VirtualBox 使用的。应用程序像往常一样，直接针对硬盘进行操作，而在底层，虚拟机将这些硬件访问转化为对文件的读写。

为了在处理器加电或者复位之后能够执行我们写的程序，势必要将这些程序写到硬盘的主引导扇区里，也就是0 面0 道1 扇区，即使是在虚拟机工作环境中，也是这样。

为了做到这一点，需要一个专门针对虚拟硬盘进行读写的工具。我自己写了一个，就在配书源代码和工具里，名叫FixVhdWr。

FixVhdWr 只针对固定尺寸的VHD。当它启动之后，首先需要选择要读写的VHD 文件。如图4-8 所示，一旦这是个合法的VHD 文件，它将读取该文件的结尾，并显示该虚拟硬盘的信息。

注意，因为FixVhdWr 只针对固定尺寸的VHD，所以，如果它检测到该VHD 是一个动态虚拟硬盘，则“下一步”按钮处于禁止状态。

第二步是选择要写入虚拟硬盘的数据文件。毕竟，在任何操作系统中，数据都是以文件的方式组织的，如图4-9 所示。

最后一个界面，是执行写入操作，如图4-10 所示，你应该选择第一种写入方式，即“LBA 连续直写模式”，并指定起始的逻辑扇区号。



图4-8　打开VHD 文件并显示该虚拟硬盘的信息

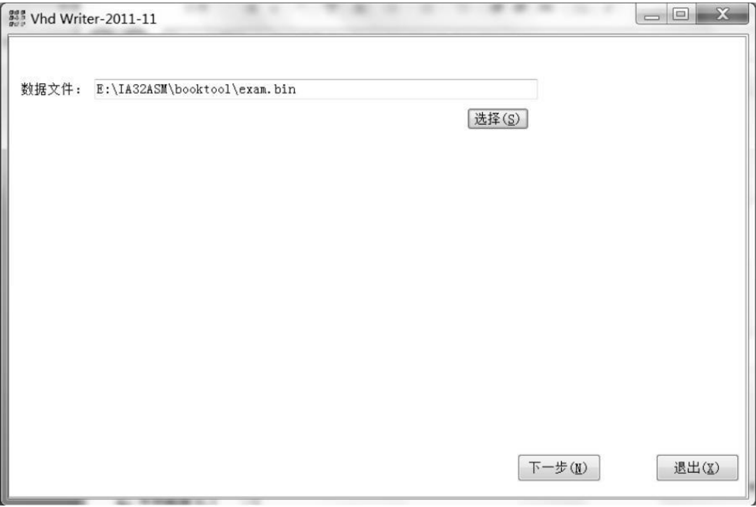


图4-9　选择要写入虚拟硬盘的文件



图4-10　指定数据写入时的起始逻辑扇区号

通常，一个扇区的尺寸是512 字节，可以看成一个数据块。所以，从这个意义上来说，硬盘是一个典型的块（Block）设备。

采用磁头、磁道和扇区这种模式来访问硬盘的方法称为CHS 模式，但不是很方便。想想看，如果有一大堆数据要写，还得注意磁头号、磁道号和扇区号不要超过界限。所以，后来引入了逻辑块地址（Logical Block Address，LBA）的概念。现在市场上销售的硬盘，无论是哪个厂家生产的，都支持LBA 模式。

LBA 模式是由硬盘控制器在硬件一级上提供支持，所以效率很高，兼容性很好。LBA 模式不考虑扇区的物理位置（磁头号、磁道号），而是把它们全部组织起来统一编号。在这种编址方式下，原先的物理扇区被组织成逻辑扇区，且都有唯一的逻辑扇区号。

比如，某硬盘有6 个磁头，每面有1000 个磁道，每磁道有17 个扇区。那么：

逻辑0 扇区对应着0 面0 道1 扇区；

逻辑1 扇区对应着0 面0 道2 扇区；

……

逻辑16 扇区对应着0 面0 道17 扇区；

逻辑17 扇区对应着1 面0 道1 扇区；

逻辑18 扇区对应着1 面0 道2 扇区；

……

逻辑33 扇区对应着1 面0 道17 扇区；

逻辑34 扇区对应着2 面0 道1 扇区；

逻辑35 扇区对应着2 面0 道2 扇区；

……

要注意到，扇区在编号时，是以柱面为单位的。即，先是0 面0 道，接着是1 面0 道，直到把所有盘面上的0 磁道处理完，再接着处理下一个柱面。之所以这样做，是因为我们讲过，要加速硬盘的访问速度，最好是尽可能不移动磁头。

因为这里总共有102000 个扇区，故最后一个逻辑扇区的编号是101999，它对应着5 面999 道17 扇区，这也是整个硬盘上最后一个物理扇区。

这里面的计算方法是：

$$\text{LBA} = C\times\text{磁头总数}\times\text{每道扇区数}+H\times\text{每道扇区数}+(S-1)$$

这里，LBA 是逻辑扇区号，C、H、S 是想求得逻辑扇区号的那个物理扇区所在的磁道、磁头和扇区号。

采用LBA 模式的好处是简化了程序的操作，使得程序员不用关心数据在硬盘上的具体位置。对于本书来说，VHD 文件是按LBA 方式组织的，一开始的512 字节就是逻辑0 扇区，然后是逻辑1 扇区；最后一个逻辑扇区排在文件的最后（最后512 个字节除外，那是VHD 文件的标识部分）。

检测点4.2

1． 运行NASMIDE 程序，输入以下汇编指令并保存为文件4-2.asm（不要考虑这些指令的含义和功能）：

```
mov ax,0xb800
mov ds,ax
mov [0x00],’a’
mov [0x02],’s’
```

```
mov [0x04],'m'

jmp $
```

2．将上面的4-2.asm 文件编译，得到二进制文件4-2.bin，并写入虚拟硬盘的主引导扇区。注意，该虚拟硬盘应当是VirtualBox 虚拟机的启动硬盘。

3．启动你的VirtualBox 虚拟机。当虚拟机启动时，会像真实的计算机一样加载硬盘上的主引导扇区代码，并执行。此时，注意观察屏幕上都显示了什么内容。

[1] 比如，当你按下主机箱面板上的RESET 按钮时，就会导致RESET 引脚电平的变化，从而使计算机热启动。

第2部分　实模式

用5章的篇幅，从多个角度展现8086处理器分段内存访问的特点，彻底理解分段的本质。

学习过程调用、栈、中断和外围设备访问的技术。

了解操作系统加载用户程序并实施重定位的一般原理。

学会用Bochs虚拟机调试程序。

第5章　编写主引导扇区代码

在学习汇编语言程序设计时，如果结合具体的实例来学习，把汇编技术融入解决一些具体的问题当中，将能获得很好的学习效果。

初学者在写第一个程序时，都有一种在屏幕上显示点什么的想法，这是很正常的，可以理解，因为屏幕是最直观的，能够看出程序的运行是否正常，是否符合设计时的预期。为此，本章将带你了解如何控制显卡在屏幕上显示字符。当然，这并不是主要目的，真正的目的在于用这个具体的实例，让你学习到以下知识：

1．NASM 汇编语言源程序的一般组成部分，如标号、指令、伪指令和注释等。

2．进一步学习mov 指令和jmp 指令的更多用法，以及加法指令add、除法指令div 和异或指令xor 的用法。

3．处理器的工作是取指令、执行指令，包括数据访问。而这一切，都是通过分段机制来完成的。在本章中，通过编写程序、分析程序的执行过程，观察程序的执行结果，进一步加深对内存分段访问机制的感性认识和对处理器工作过程的理解。

5.1 本章代码清单

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：5-1（主引导扇区程序）

源程序文件：c05_mbr.asm

5.2 欢迎来到主引导扇区

在前面的预备知识里，我们已经知道，处理器加电或者复位之后，如果硬盘是首选的启动设备，那么，ROM-BIOS 将试图读取硬盘的0 面0 道1 扇区。传统上，这就是主引导扇区（Main Boot Sector，MBR）。

读取的主引导扇区数据有512 字节，ROM-BIOS 程序将它加载到逻辑地址0x0000:0x7c00处，也就是物理地址0x07c00 处，然后判断它是否有效。

一个有效的主引导扇区，其最后两字节应当是0x55 和0xAA。ROM-BIOS 程序首先检测这两个标志，如果主引导扇区有效，则以一个段间转移指令jmp 0x0000:0x7c00 跳到那里继续执行。

一般来说，主引导扇区是由操作系统负责的。正常情况下，一段精心编写的主引导扇区代码将检测用来启动计算机的操作系统，并计算出它所在的硬盘位置。然后，它把操作系统的自举代码加载到内存，也用jmp 指令跳转到那里继续执行，直到操作系统完全启动。

在本章中，我们将试图写一段程序，把它编译之后写入硬盘的主引导扇区，然后让处理器执行。当然，仅仅执行还不够，还必须在屏幕上显示点什么，要不然的话，谁知道我们的程序是不是成功运行了呢？

通过本章的学习，我们可以对处理器如何执行指令、如何访问内存以及如何进行算术逻辑运算有一个最基本的认知。

5.3 注 释

如本章代码清单5-1 所展示的那样，在汇编语言源程序里，注释用于说明本程序的用途和编写时间等，可以单独成行，也可以放在每条指令的后面，解释本指令的目的和功能。注释不但有助于其他编程人员理解当前程序的编写思路和工作原理，而且也能帮助你自己在以后的某个时间重拾这些记忆。

注释必须以英文字母“;”开始。

在源程序编译阶段，编译器将忽略所有注释。因此，在编译之后，这些和生成机器代码无关的内容都统统消失了。

5.4　在屏幕上显示文字

5.4.1　显卡和显存

本程序首先要做的事是在屏幕上显示一行文字。当然，要想在屏幕上显示文字，就需要先了解文字是如何显示在屏幕上的。

为了显示文字，通常需要两种硬件，一是显示器，二是显卡。显卡的职责是为显示器提供内容，并控制显示器的显示模式和状态，显示器的职责是将那些内容以视觉可见的方式呈现在屏幕上。

一般来说，显卡都是独立生产、销售的部件，需要插在主板上才能工作。当然，像处理器、内存这样的东西，也位于主板上。每台计算机都有主板，它就在机箱内部，有时间你可以打开机箱来观察一下。

当然，显卡未必一定是独立的插卡。为了节省使用者的成本，有的显卡会直接做在主板上，这样的显卡也有个名字，叫集成显卡。

显卡控制显示器的最小单位是像素，一个像素对应着屏幕上的一个点。屏幕上通常有数十万乃至更多的像素，通过控制每个像素的明暗和颜色，我们就能让这大量的像素形成文字和美丽的图像。

不过，一个很容易想到的问题是，如何来控制这些像素呢？

答案是显卡都有自己的存储器，因为它位于显卡上，故称显示存储器（Video RAM：VRAM），简称显存，要显示的内容都预先写入显存。和其他半导体存储器一样，显存并没有什么特殊的地方，也是一个按字节访问的存储器件。

对显示器来说，显示黑白图像是最简单的，因为只需要控制每个像素是亮，还是不亮。如果把不亮当成比特“0”，亮看成比特“1”，那就好办了。因为，只要将显存里的每个比特和显示器上的每个像素对应起来，就能实现这个目标。

如图5-1 所示，显存的第1 个字节对应着屏幕左上角连续的8 个像素；第2 个字节对应着屏幕上后续的8 个像素，后面的以此类推。

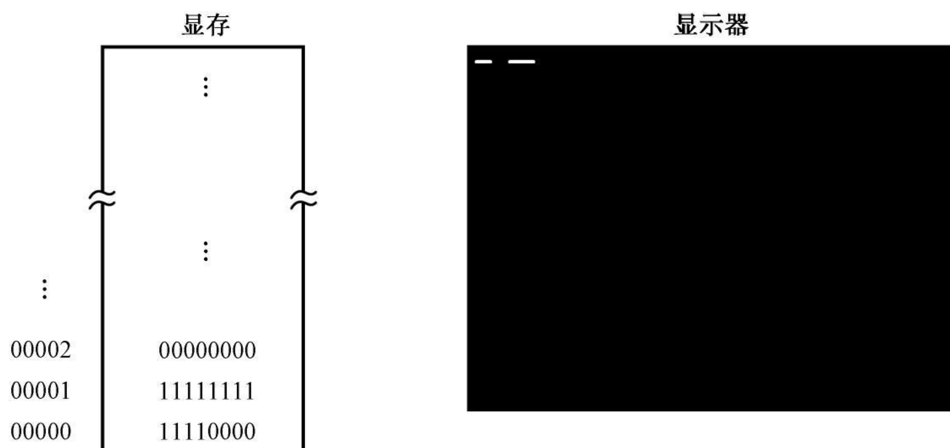


图5-1　显存内容和显示器内容之间的对应关系

显卡的工作是周期性地从显存中提取这些比特，并把它们按顺序显示在屏幕上。如果是比特“0”，则像素保持原来的状态不变，因为屏幕本来就是黑的；如果是比特“1”，则点亮对应的像素。

继续观察图5-1，假设显存中，第1 个字节的内容是11110000，第2 个字节的内容是11111111，其他所有的字节都是00000000。在这种情况下，屏幕左上角先是显示4 个亮点，再显示4 个黑点，然后再显示8 个亮点。因为像素是紧挨在一起的，所以我们看到的先是一条白短线，隔着一定距离（4 个像素）又是一条白长线。

黑色和白色只需要1 个比特就能表示，但要显示更多的颜色，1 个比特就不够了。现在最流行的，是用24 个比特，即3 个字节，来对应一个像素。因为224=16777216，所以在这种模式下，同屏可以显示16777216 种颜色，这称为真彩色。有关颜色的显示和它们与字长的关系，在《穿越计算机的迷雾》一书中有详细的介绍，这里不再赘述。

上面所讨论的，是人们常说的图形模式。图形模式是最容易理解的，同时对显示器来说也是最自然的模式。

现在是图形的时代，就连手机的屏幕都是五彩缤纷的。时光倒退到几十年前，在那个时代，真彩色还没有出现，显示器只能提供有限的色彩，处理器也不够强劲（以今天的眼光来看）。在这种情况下，人们不太可能认为图形显示技术有多么重要，因为他们不看高清电影，也没有数码相机，用计算机制作动画片更是不能想象的事。那个时候，人们的愿望很简单，只要能显示文字就行。

不管是显示图片，还是文字，对显示器来说没有什么不同，因为所有的内容都是由像素组成的，区别仅仅在于这些像素组成的是什么。有时候，人们会说，哦，显示的是一棵树；有时候，人们会说，哦，显示的是一个字母“H”。

问题是，操作显存里的比特，使得屏幕上能显示出字符的形状，是非常麻烦、非常繁重的工作，因为你必须计算该字符所对应的比特位于显存里的什么位置。

为了方便，工程师们想出了一个办法。就像一个二进制数既可以是一个普通的数，也可以代表一条处理器指令一样，他们认为每个字符也可以表示成一个数。比如，数字0x4C 就代表字符“L”，这个数被称为是字符“L”的ASCII 代码，后面会讲到。

如图5-2 所示，可以将字符的代码存放到显存里，第1 个代码对应着屏幕左上角第1 个字符，第2 个代码对应着屏幕左上角第2 个字符，后面的以此类推。剩下的工作是如何用代码来控制屏幕上的像素，使它们或明或暗以构成字符的轮廓，这是字符发生器和控制电路的事情。

传统上，这种专门用于显示字符的工作方式称为文本模式。文本模式和图形模式是显卡的两种基本工作模式，可以用指令访问显卡，设置它的显示模式。在不同的工作模式下，显卡对显存内容的解释是不同的。

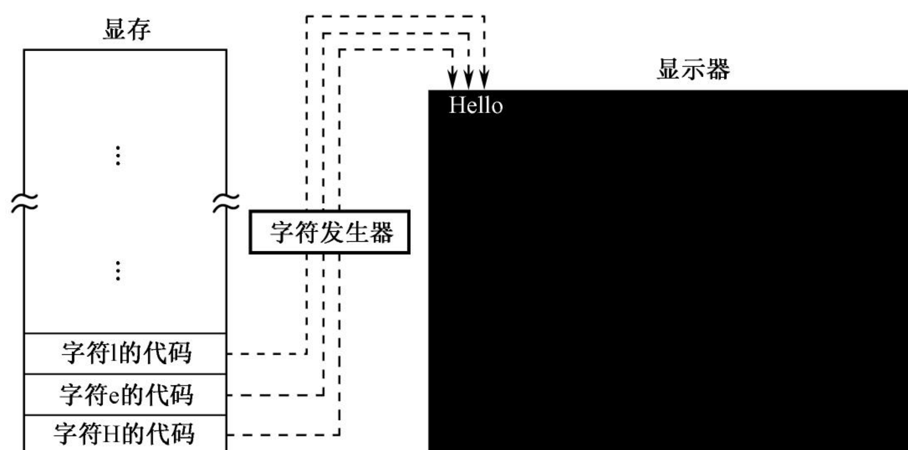


图5-2　字符在屏幕上的显示原理

为了给出要显示的字符，处理器需要访问显存，把字符的ASCII 码写进去。但是，显存是位于显卡上的，访问显存需要和显卡这个外围设备打交道。同时，多一道手续自然是不好的，这当中最重要的考量是速

度和效率。想想看，你让人传话给父母，和自己亲自往家里打电话，花费的时间是不一样的。为了实现一些快速的游戏动画效果，或者播放高码率的电影，不直接访问显存是办不到的。

为此，计算机系统的设计者们，这些敢想敢干的人，决定把显存映射到处理器可以直接访问的地址空间里，也就是内存空间里。

如图5-3 所示，我们知道，8086 可以访问1MB 内存。其中，0x00000～9FFFF 属于常规内存，由内存条提供；0xF0000～0xFFFFF 由主板上的一个芯片提供，即ROM-BIOS。

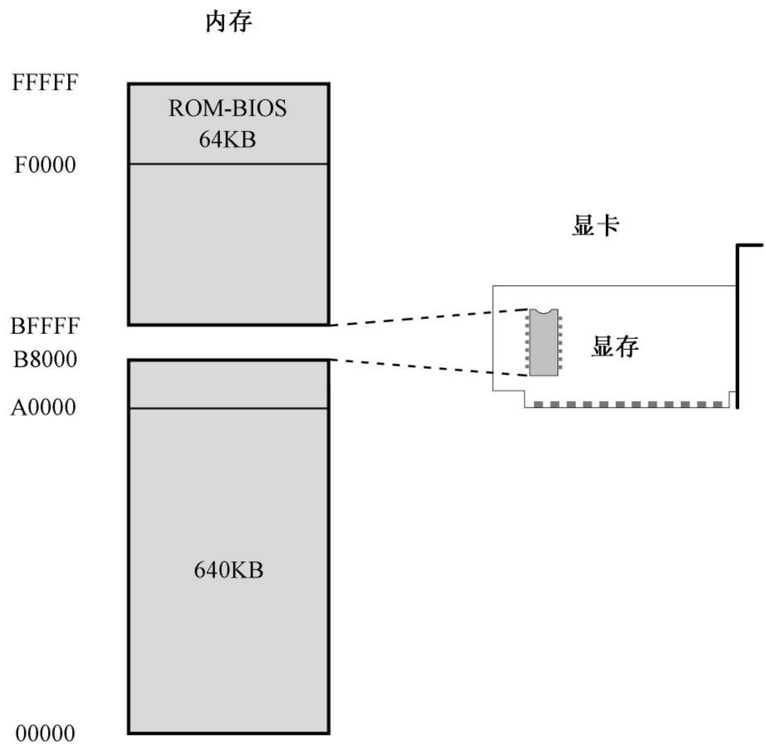


图5-3　文本模式下显存到内存的映射

这样一来，中间还有一个320KB 的空洞，即0xA0000～0xEFFFF。传统上，这段地址空间由特定的外围设备来提供，其中就包括显卡。因为显示功能对于现代计算机来说实在是太重要了。

由于历史的原因，所有在个人计算机上使用的显卡，在加电自检之后都会把自己初始化到80× 25 的文本模式。在这种模式下，屏幕上可以显示25 行，每行80 个字符，每屏总共2000 个字符。

所以，如图5-3 所示，一直以来，0xB8000～0xBFFFF 这段物理地址空间，是留给显卡的，由显卡来提供，用来显示文本。除非显卡出了

毛病，否则这段空间总是可以访问的。如果显卡出了毛病怎么办呢？很简单，计算机一定不会通过加电自检过程，这就是传说中的严重错误，计算机是无法启动的，更不要说加载并执行主引导扇区的内容了。

5.4.2　初始化段寄存器

和访问主内存一样，为了访问显存，也需要使用逻辑地址，也就是采用“段地址：偏移地址”的形式，这是处理器的要求。考虑到文本模式下显存的起始物理地址是0xB8000，这块内存可以看成是段地址为0xB800，偏移地址从0x0000 延伸到0xFFFF 的区域，因此我们可以把段地址定为0xB800。

访问内存可以使用段寄存器DS，但这不是强制性的，也可以使用ES。因为DS 还有别的用处，所以在这里我们使用ES 来指向显存所在的段。

源程序第6、7 行，首先把立即数0xB800 传送到AX，然后再把AX的值传送到ES。这样，附加段寄存器ES 就指向0xB800 段（段基地址为0xB800）。

你可能会想，为什么不直接这样写：

```
mov es,0xb800
```

而要用寄存器AX 来中转呢？

原因是不存在这样的指令，Intel 的处理器不允许将一个立即数传送到段寄存器，它只允许这样的指令：

```
mov 段寄存器，通用寄存器
mov 段寄存器，内存单元
```

没有人能够说清楚这里面的原因，Intel 公司似乎也从没有提到过这件事，尽管从理论上，这是可行的。我们只能想，也许Intel 是出于好心，避免我们无意中犯错，毕竟，段地址一旦改变，后面对内存的访问都会受到影响。理论上，麻烦一点的方法，可以保证你确实知道自己在做什么。

5.4.3 显存的访问和ASCII 代码

一旦将显存映射到处理器的地址空间，那么，我们就可以使用普通的传送指令（mov）来读写它，这无疑是非常方便的，但需要首先将它作为一个段来看待，并将它的基地址传送到段寄存器。

为此，源程序的第10、11 行，我们把0xB800 作为段地址传送到附加段寄存器ES，以后就用ES 来读写显存。这样，段内偏移为0 的位置就对应着屏幕左上角的字符。

在计算机中，每个用来显示在屏幕上的字符，都有一个二进制代码。这些代码和普通的二进制数字没有什么不同，唯一的区别在于，发送这些数字的硬件和接收这些数字的硬件把它们解释为字符，而不是指令或者用于计算的数字。

这就是说，在计算机中，所有的东西都是无差别的数字，它们的意义，只取决于生成者和使用者之间的约定。为了在终端和大型主机，以及主机和打印机、显示器之间交换信息，1967 年，美国国家标准学会制定了美国信息交换标准代码（American Standard Code for Information Interchange，ASCII），如表5-1 所示。

表5-1 ASCII 表

$b_6b_5b_4$ / $b_3b_2b_1b_0$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

在不同设备之间，或者在同一设备的不同模块之间有一个信息传递标准是非常必要的。想想看，当你用手机向朋友发送短消息时，这些文字当然被编码成二进制数字。如果对方的手机使用了不同的编码，那么他将无法正确还原这些消息，而很可能显示为乱码。

值得注意的是，ASCII 是7 位代码，只用了一个字节中的低7 比特，最高位通常置0。这意味着，ASCII 只包含128 个字符的编码。所以，在表中，水平方向给出了代码的高3 比特，而垂直方向给出了代码的低4 比特。比如字符“*”，它的代码是二进制数的010 1010，即0x2A。

ASCII 表中有相当一部分代码是不可打印和显示的，它们用于控制通信过程。比如，LF 是换行；CR 是回车；DEL 和BS 分别是删除和退格，在我们平时用的键盘上也是有的；BEL 是振铃（使远方的终端响铃，以引起注意）；SOH 是文头；EOT 是文尾；ACK 是确认，等等。

注意，一定要遵从约定。比如，你在处理器上编写程序算了一道数学题2+3，你也希望把结果5 显示在屏幕上。这个时候，算出的结果是0000 0101，即0x05。但是，数字5 和字符5 是不同的，显卡在任何时候都认为你发送的是ASCII 码。所以，你不应该发送0x05，而应该发送0x35。

屏幕上的每个字符对应着显存中的两个连续字节，前一个是字符的ASCII 代码，后面是字符的显示属性，包括字符颜色（前景色）和底色（背景色）。如图5-4 所示，字符“H”的ASCII 代码是0x48，其显示属性是0x07；字符“e”的ASCII 代码是0x65，其显示属性是0x07。

如图5-4 所示，字符的显示属性（1 字节）分为两部分，低4 位定义的是前景色，高4 位定义的是背景色。色彩主要由R、G、B 这3 位决定，毕竟我们知道，可以由红（R）、绿（G）、蓝（B）三原色来配出其他所有颜色。K 是闪烁位，为0 时不闪烁，为1 时闪烁；I 是亮度位，为0时正常亮度，为1 时呈高亮。表5-2 给出了背景色和前景色的所有可能值。

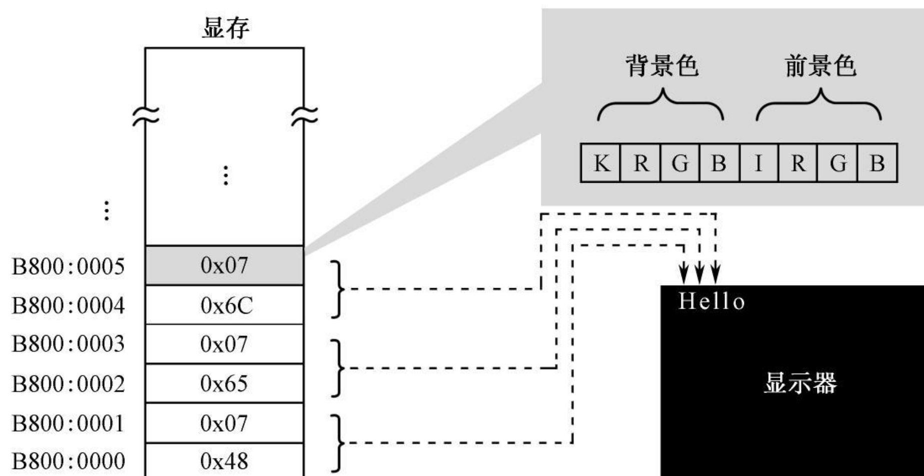


图5-4　字符代码及字符属性示意图

表5-2　80×25 文本模式下的颜色表

R	G	B	背景色	前景色	
			K=0 时不闪烁，K=1 时闪烁	I=0	I=1
0	0	0	黑	黑	灰
0	0	1	蓝	蓝	浅蓝
0	1	0	绿	绿	浅绿
0	1	1	青	青	浅青
1	0	0	红	红	浅红
1	0	1	品（洋）红	品（洋）红	浅品（洋）红
1	1	0	棕	棕	黄
1	1	1	白	白	亮白

从表5-2 来看，图5-4 中的字符属性0x07 可以解释为黑底白字，无闪烁，无加亮。

你可能觉得奇怪，当屏幕上一片漆黑，什么内容都没有的时候，显存里会是什么内容呢？

实际上，这个时候，屏幕上显示的全是黑底白字的空白字符，也叫空格字符（Space），ASCII代码是0x20，当你用大拇指按动键盘上最长的那个键时，就产生这个字符。因为它是空白，自然就无法在黑底上看到任何痕迹了。

5.4.4　显示字符

从源程序的第10 行开始，到第35 行，目的是显示一串字符“Label offset:”。为此，需要把它们每一个的ASCII 码顺序写到显存中。

为了方便，多数汇编语言编译器允许在指令中直接使用字符的字面值来代替数值形式的ASCII 码，比如源程序第10 行：

```
mov byte [es:0x00],'L'
```

这等效于

```
mov byte [es:0x00],0x4c
```

尽管通过查表可以知道字符“L”的ASCII 代码是0x4C，但毕竟费事。不过，要在指令中使用字符的字面值，这个字符必须用引号围起来，就像上面一样。在源程序的编译阶段，汇编语言编译器会将它转换成ASCII 码的形式。

当前的mov 指令是将立即数传送到内存单元，目的操作数是内存单元，源操作数是立即数（ASCII 代码）。为了访问内存单元，需要给出段地址和偏移地址。在这条指令中，偏移地址0x00，段地址在哪里呢？一般情况下，如果没有附加任何指示，段地址默认在段寄存器DS 中。比如：

```
mov byte [0x00],’L’
```

当执行这条指令后，处理器把段寄存器DS 的内容左移4 位（相当于乘以十进制数16 或者十六进制数0x10），加上这里的偏移地址0x00，就得到了物理地址。

但是实际上，在我们的程序中，显存的段地址位于段寄存器ES 中，我们希望使用ES 来访问内存。因此，这里使用了段超越前缀“es:”。这就是说，我们明确要求处理器在生成物理地址时，使用段寄存器ES，而不是默认情况下的DS。

因为指令中给出的偏移地址是0x00，且ES 的值已经在前面被设为0xB800，故它指向ES 段中，偏移地址为0 的内存单元，即0xB800:0x0000，也就是物理地址0xB8000，这个内存单元对应着屏幕左上角第一个字符的位置。

还需要注意的是，因为目的操作数给出的是一个内存地址，我们要用源操作数来修改这个地址里的内容，所以，目的操作数必须用方括号围起来，以表明它是一个地址，处理器应该用这个地址再次访问内存，将源操作数写进这个单元。实际上，这类似于高级语言里的指针。

最后，关键字“byte”用来修饰目的操作数，指出本次传送是以字节的方式进行的。在16 位的处理器上，单次操作的数据宽度可以是8 位，也可以是16 位。到底是8 位，还是16 位，可以根据目的操作数或者源操作数来判断。遗憾的是，在这里，目的操作数是偏移地址0x00，它可以是字节单元，也可以是字单元，到底是哪一种，无法判断；而源操作数呢，是立即数0x4C，它既可以解释为8 位的0x4C，也可以解释为16 位的0x004C。在这种情况下，编译器将无法搞懂你的真实意图，只能报告错误，所以必须用“byte”或者“word”进行修饰（明确指示）。于是，一旦目的操作数被指明是“byte”的，那么，源操作数的宽度也就明确了。相反地，下面的指令就不需要任何修饰：

```
mov [0x00],al        ;按字节操作
mov al,[0x02]        ;按字操作
```

因为屏幕上的一个字符对应着内存中的两个字节：ASCII 代码和属性，所以，源程序第11 行的功能是将属性值0x07 传送到下一个内存单元，即偏移地址0x01 处。这个属性可以解释为黑底白字，无闪烁，也无加亮，请参阅表5-2。

后面，从第12 行开始，到第35 行，用于向显示缓冲区填充剩余部分的字符。注意，在这个过程中，偏移地址一直是递增的。

5.4.5 MOV 指令的格式

到目前为止，我们已经多次接触了mov 指令。在处理器的整个指令集中，mov 指令是用得最多的一条。

mov 指令用于数据传送。既然是数据传送，那么，目的操作数的作用应该相当于一个“容器”，故必须是通用寄存器或者内存单元；源操作数呢，也可以是和目的操作数具有相同数据宽度的通用寄存器和内存单元，还可以是立即数。传送指令只影响目的操作数的内容，不改变源操作数的内容。比如：

```
mov ah,bh
mov ax,dx
```

以上，第一条指令的目的操作数和源操作数都是8 位寄存器，指令执行后，寄存器AH 的内容和BH 相同；第二条指令的目的操作数和源操作数都是16 位寄存器，指令执行后，寄存器AX的内容和DX 相同。但是，由于数据宽度不同，下面这条指令就是错误的：

```
mov ax,bl
```

再来看下面两条指令：

```
mov [0x02],bh
mov ax,[0x06]
```

以上，第一条指令是把寄存器BL 中的内容传送到偏移地址为0x02 的8 位内存单元；第二条指令是把偏移地址为0x06 的16 位内存单元里的内容传送到寄存器AX 中。由于这两条指令中都有寄存器操作数，故不需要用“byte”或者“word”来修饰。

传送指令的源操作数也可以是立即数。比如：

```
mov ah,0x05
mov word [0x1c],0xf000
```

以上，第一条指令是把立即数0x05 传送到寄存器AH 中，指令执行后，AH 中的内容为0x05；第二条指令是把立即数0xf000 传送到偏移地址为0x1c 的16 位内存单元中。因为上一节所说的原因，这里要用word 来修饰。

mov 指令的目的操作数不允许为立即数，而且，目的操作数和源操作数不允许同时为内存单元。因此，下面两条指令都是不正确的：

```
mov 0x1c,al
mov [0x01],[0x02]
```

以上，说第一条指令是错误的，这很好理解。想想看，你把寄存器AL 中的内容传送给一个立即数，这是什么意思呢？于理不通。至于第二条指令为什么不正确，那是因为处理器不允许在两个内存单元之间直接

进行传送操作。事实上，这条指令的功能可以用两条指令实现（假设传送的是一个字）：

```
mov ax,[0x02]
mov [0x01],ax
```

就算处理器支持在两个内存单元之间直接传送数据，那么，它依然是在内部按上面的两个步骤进行操作的。而且，支持这种直接传送操作还需要增加额外的电路。

不单单是mov 指令，其他指令都不支持在两个内存单元之间直接进行操作，包括加、减、乘、除和逻辑运算等指令。事情是明摆着的，既然增加了处理器的复杂性之后和用两条指令没什么区别，干脆就用两条指令好了。

检测点5.1

1． 在我们日常使用的个人计算机上，文本模式下的显示缓冲区被映射到物理内存地址空间，起始地址为（ ），它对应的段地址为（ ）。在标准的80×25 文本模式下，要想在屏幕右下角显示一个绿底白字的字符“H”，那么，应当在该段内偏移量为（ ）的地方开始，连续写入两个字节（ ）和（ ）。

2． 以下指令中，哪些是正确的，不正确的原因是什么？

A.mov al,0x55aa B.mov ds,0x6000 C.mov ds,al

D.mov [0x06],0x55aa E.mov ds,bx F.mov ax,0x02

G.mov word [0x0a],ax H.mov es,cx I.mov ax,bl

J.mov byte [0x00],’c’ K.mov [0x02],[0xf000] L.mov ds,[0x03]

5.5 显示标号的汇编地址

5.5.1 标号

处理器访问内存时，采用的是“段地址：偏移地址”的模式。对于任何一个内存段来说，段地址可以开始于任何16 字节对齐的地方，偏移地址则总是从0x0000 开始递增。

为了支持这种内存访问模式，在源程序的编译阶段，编译器会把源程序5-1 整体上作为一个独立的段来处理，并从0 开始计算和跟踪每一条指令的地址。因为该地址是在编译期间计算的，故称为**汇编地址**。汇编地址是在源程序编译期间，编译器为每条指令确定的汇编位置（Assembly Position），指示该指令相对于程序或者段起始处的距离，以字节计。当编译后的程序装入物理内存后，它又是该指令在内存段内的偏移地址。

如表5-3 所示，在用我们的配书工具Nasmide 书写并编译代码清单5-1 后，除了生成一个以“.bin”为扩展名的二进制文件，还会生成一个以“.lst”为扩展名的列表文件。这张表列出的，就是本章代码清单5-1 编译后生成的列表文件内容。

表5-3 共分五栏，从左到右依次是行号、指令的汇编地址、指令编译后的机器代码、源程序代码和注释。可以看出，第一条指令mov ax,0xb800 的汇编地址是0x00000000,对应的机器代码为B8 00 B8；第二条指令mov es,ax 的汇编地址是0x00000003，机器代码为8E C0。

表5-3　代码清单5-1 编译后的列表文件内容

```
1                                          ;代码清单 5-1
2                                          ;文件名：c05_mbr.asm
3                                          ;文件说明：硬盘主引导扇区代码
4                                          ;创建日期：2011-3-31 21:15
5
6       00000000    B800B8                 mov ax,0xb800                 ;指向文本模式的显示缓冲区
7       00000003    8EC0                   mov es,ax
8
9                                          ;以下显示字符串"Label offset:"
10      00000005    26C60600004C           mov byte [es:0x00],'L'
11      0000000B    26C606010007           mov byte [es:0x01],0x07
12      00000011    26C606020061           mov byte [es:0x02],'a'
13      00000017    26C606030007           mov byte [es:0x03],0x07
14      0000001D    26C606040062           mov byte [es:0x04],'b'
15      00000023    26C606050007           mov byte [es:0x05],0x07
16      00000029    26C606060065           mov byte [es:0x06],'e'
17      0000002F    26C606070007           mov byte [es:0x07],0x07
18      00000035    26C60608006C           mov byte [es:0x08],'l'
19      0000003B    26C606090007           mov byte [es:0x09],0x07
20      00000041    26C6060A0020           mov byte [es:0x0a],' '
21      00000047    26C6060B0007           mov byte [es:0x0b],0x07
```

```
22   0000004D   26C6060C006F   mov byte [es:0x0c],"o"
23   00000053   26C6060D0007   mov byte [es:0x0d],0x07
24   00000059   26C6060E0066   mov byte [es:0x0e],'f'
25   0000005F   26C6060F0007   mov byte [es:0x0f],0x07
26   00000065   26C606100066   mov byte [es:0x10],'f'
27   0000006B   26C606110007   mov byte [es:0x11],0x07
28   00000071   26C606120073   mov byte [es:0x12],'s'
29   00000077   26C606130007   mov byte [es:0x13],0x07
30   0000007D   26C606140065   mov byte [es:0x14],'e'
31   00000083   26C606150007   mov byte [es:0x15],0x07
32   00000089   26C606160074   mov byte [es:0x16],'t'
33   0000008F   26C606170007   mov byte [es:0x17],0x07
34   00000095   26C60618003A   mov byte [es:0x18],':'
35   0000009B   26C606190007   mov byte [es:0x19],0x07
36
37   000000A1   B8[2E01]       mov ax,number                  ;取得标号 number 的偏移地址
38   000000A4   BB0A00         mov bx,10
39
40                             ;设置数据段的基地址
41   000000A7   8CC9           mov cx,cs
42   000000A9   8ED9           mov ds,cx
43
44                             ;求个位上的数字
45   000000AB   BA0000         mov dx,0
46   000000AE   F7F3           div bx
47   000000B0   8816[2E7D]     mov [0x7c00+number+0x00],dl    ;保存个位上的数字
48
49                             ;求十位上的数字
50   000000B4   31D2           xor dx,dx
51   000000B6   F7F3           div bx
52   000000B8   8816[2F7D]     mov [0x7c00+number+0x01],dl    ;保存十位上的数字
53
54                             ;求百位上的数字
55   000000BC   31D2           xor dx,dx
56   000000BE   F7F3           div bx
57   000000C0   8816[307D]     mov [0x7c00+number+0x02],dl    ;保存百位上的数字
58
59                             ;求千位上的数字
60   000000C4   31D2           xor dx,dx
61   000000C6   F7F3           div bx
62   000000C8   8816[317D]     mov [0x7c00+number+0x03],dl    ;保存千位上的数字
63
64                             ;求万位上的数字
```

65	000000CC	31D2	xor dx,dx
66	000000CE	F7F3	div bx
67	000000D0	8816[327D]	mov [0x7c00+number+0x04],dl ;保存万位上的数字
68			
69			;以下用十进制显示标号的偏移地址
70	000000D4	A0[327D]	mov al,[0x7c00+number+0x04]
71	000000D7	0430	add al,0x30
72	000000D9	26A21A00	mov [es:0x1a],al
73	000000DD	26C6061B0004	mov byte [es:0x1b],0x04
74			
75	000000E3	A0[317D]	mov al,[0x7c00+number+0x03]
76	000000E6	0430	add al,0x30
77	000000E8	26A21C00	mov [es:0x1c],al
78	000000EC	26C6061D0004	mov byte [es:0x1d],0x04
79			
80	000000F2	A0[307D]	mov al,[0x7c00+number+0x02]
81	000000F5	0430	add al,0x30
82	000000F7	26A21E00	mov [es:0x1e],al
83	000000FB	26C6061F0004	mov byte [es:0x1f],0x04
84			
85	00000101	A0[2F7D]	mov al,[0x7c00+number+0x01]
86	00000104	0430	add al,0x30
87	00000106	26A22000	mov [es:0x20],al
88	0000010A	26C606210004	mov byte [es:0x21],0x04
89			
90	00000110	A0[2E7D]	mov al,[0x7c00+number+0x00]
91	00000113	0430	add al,0x30
92	00000115	26A22200	mov [es:0x22],al
93	00000119	26C606230004	mov byte [es:0x23],0x04
94			
95	0000011F	26C606240044	mov byte [es:0x24],'D'
96	00000125	26C606250007	mov byte [es:0x25],0x07
97			
98	0000012B	E9FDFF	infi: jmp near infi ;无限循环
99			
100	0000012E	0000000000	number db 0,0,0,0,0
101			
102	00000133	00<rept>	times 203 db 0
103	000001FE	55AA	db 0x55,0xaa

从表5-3 中可以看出，在编译阶段，每条指令都被计算并赋予了一个汇编地址，就像它们已经被加载到内存中的某个段里一样。实际上，如

图5-5 所示，当编译好的程序加载到物理内存后，它在段内的偏移地址和它在编译阶段的汇编地址是相同的。

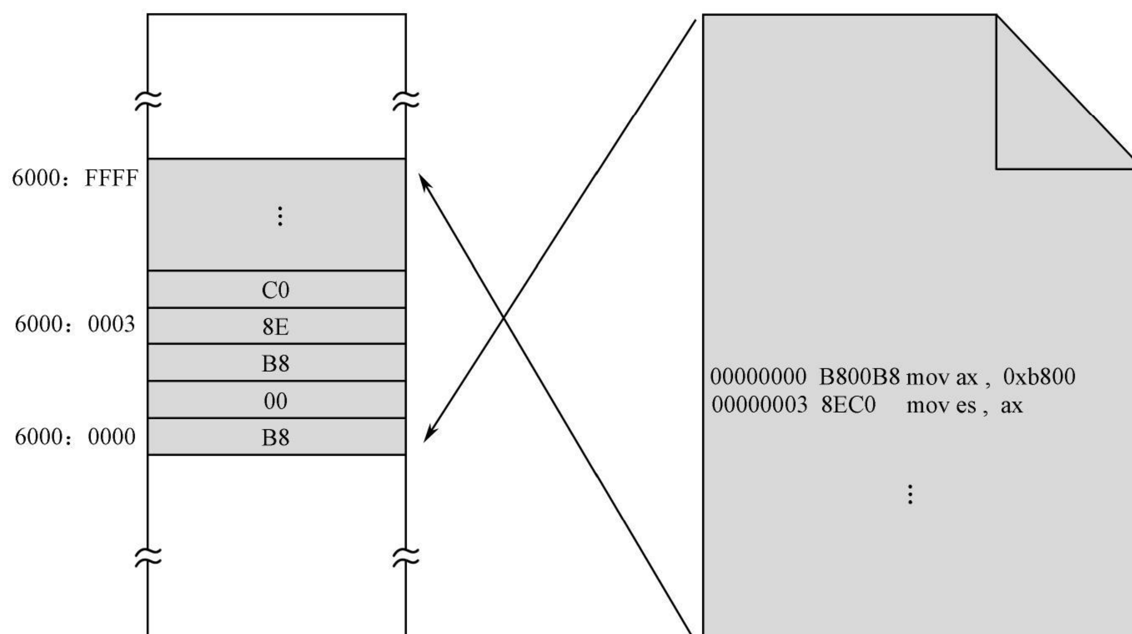


图5-5　汇编地址和偏移地址的关系

正如图5-5 所示，编译后的程序是整体加载到内存中某个段的，交叉箭头用于指示它们之间的映射关系。之所以箭头是交叉的，是因为源程序的编译是从上往下的，而内存地址的增长是从下往上的（从低地址往高地址方向增长）。

图5-5 中假定程序是从内存物理地址0x60000 开始加载的。因为该物理地址也对应着逻辑地址0x6000:0x0000，因此我们可以说，该程序位于段0x6000 内。

在编译阶段，源程序的第一条指令mov ax,0xb800 的汇编地址是0x00000000，而它在整个程序装入内存后，在段内的偏移地址是0x0000，即逻辑地址0x6000:0000，两者的偏移地址是一致的。

再看源程序的第二条指令，是mov es,ax，它在编译阶段的汇编地址是0x00000003。在整个程序装入内存后，它在段内的偏移地址是0x0003，也没有变化。

这就很好地说明了汇编地址和偏移地址之间的对应关系。理解这一点，对后面的编程很重要。

在NASM 汇编语言里，每条指令的前面都可以拥有一个标号，以代表和指示该指令的汇编地址。毕竟，由我们自己来计算和跟踪每条指令所在的汇编地址是极其困难的。这里有一个很好的例子，比如源程序第98 行：

```
infi: jmp near infi
```

在这里，行首带冒号的是标号是“infi”。请看表5-3，这条指令的汇编地址是0x0000012B，故infi 就代表数值0x0000012B，或者说是0x0000012B 的符号化表示。

标号之后的冒号是可选的。所以下面的写法也是正确的：

```
infi jmp near infi
```

标号并不是必需的，只有在我们需要引用某条指令的汇编地址时，才使用标号。正是因为这样，本章源程序中的绝大多数指令都没有标号。

标号可以单独占用一行的位置，像这样：

```
infi:
    jmp near infi
```

这种写法和第98 行相比，效果并没有什么不同，因为infi 所在的那一行没有指令，它的地址就是下一行的地址，换句话说，和下一行的地址是相同的。

标号可以由字母、数字、“_”、“$”、“#”、“@”、“～”、“.”、“？”组成，但必须以字母、“.”、“_”和“？”中的任意一个打头。

5.5.2 如何显示十进制数字

我们已经知道，标号代表并指示它所在位置处的汇编地址。现在，我们要编写指令，在屏幕上把这个地址的数值显示出来。为此，源程序的第37 行用于获取标号所代表的汇编地址：

```
mov ax,number
```

标号“number”位于源程序的第100 行，只不过后面没有跟着冒号“:”。你当然可以加上冒号，但这无关紧要。注意，传送到寄存器AX 的值是在源程序编译时确定的，在编译阶段，编译器会将标号number 转换成立即数。如表5-3 所示，标号number 处的汇编地址是0x012E，因此，这条语句其实就是（等效于）

```
mov ax,0x012E
```

问题在于，如果不是借助于别的工具和手段，你不可能知道此处的汇编地址是0x012E。所以，在汇编语言中使用标号的好处是不必关心这些。

因此，当这条指令编译后，得到的机器指令为B8[2E01]，或者B8 2E 01。B8 是操作码，后面是字操作数0x012E，只不过采用的是低端字节序。

十六进制数0x012E 等于十进制数302，但是，通过前面对字符显示原理的介绍，我们应该清楚，直接把寄存器AX 中的内容传送到显示缓冲区，是不可能在屏幕上出现“302”的。

解决这个问题的办法是将它的每个数位单独拆分出来，这需要不停地除以10。

考虑到寄存器AX 是16 位的， 可以表示的数从二进制的0000000000000000 到1111111111111111，也就是十进制的0～65535，故它可以容纳最大5 个数位的十进制数，从个位到万位，比如61238。那么，假如你并不知道它是多少，只知道它是一个5 位数，那么，如何通过分解得到它的每个数位呢？

首先，用61238 除以10，商为6123，余8，本次相除的余数8 就是个位数字；

然后，把上一次的商数6123 作为被除数，再次除以10，商为612，余3，余数3 就是十位上的数字；

接着，再用上一次的商数612 除以10，商为61，余2，余数2 就是百位上的数字；

同上，再用61 除以10，商为6，余1，余数1 就是千位上的数字；

最后，用6 除以10，商为0，余6，余数6 就是万位上的数字。

很显然，只要把AX 的内容不停地除以10，只需要5 次，把每次的余数反向组合到一起，就是原来的数字。同样，如果反向把每次的余数显示到屏幕上，应该就能看见这个十进制数是多少了。

不过，即使是得到了单个的数位，也还是不能在屏幕上显示，因为它们是数字，而非ASCII代码。比如，数字0x05 和字符“5”是不同的，后者实际上是数字0x35。

观察表5-1，你会发现，字符“0”的ASCII 代码是0x30，字符“1”的ASCII 代码是0x31，字符“9”的ASCII 代码是0x39。这就是说，把每次相除得到的余数加上0x30，在屏幕上显示就没问题了。

5.5.3　在程序中声明并初始化数据

可以用处理器提供的除法指令来分解一个数的各个数位，但是每次除法操作后得到的数位需要临时保存起来以备后用。使用寄存器不太现实，因为它的数量很少，且还要在后续的指令中使用。因此，最好的办法是在内存中专门留出一些空间来保存这些数位。

尽管我们的目的仅仅是分配一些空间，但是，要达到这个目的必须初始化一些初始数据来“占位”。这就好比是排队买火车票，你可以派任何无关的人去帮你占个位置，真正轮到你买的时候，你再出现。源程序的第100 行用于声明并初始化这些数据，而标号number 则代表了这些数据的起始汇编地址。

要放在程序中的数据是用DB 指令来声明（Declare）的，DB 的意思是声明字节（Declare Byte），所以，跟在它后面的操作数都占一个字节的长度（位置）。注意，如果要声明超过一个以上的数据，各个操作数之间必须以逗号隔开。

除此之外，DW（Declare Word）用于声明字数据，DD（Declare Double Word）用于声明双字（两个字）数据，DQ（Declare Quad Word）用于声明四字数据。DB、DW、DD 和DQ 并不是处理器指令，它只是编译器提供的汇编指令，所以称做伪指令（pseudo Instruction）。伪指令是汇编指令的一种，它没有对应的机器指令，所以它不是机器指令的助记符，仅仅在编译阶段由编译器执行，编译成功后，伪指令就消失了，所以在程序执行时，伪指令是得不到处理器光顾的，实际上，程序执行时，伪指令已不存在。

声明的数据可以是任何值，只要不超过伪指令所指示的大小。比如，用DB 声明的数据，不能超过一个字节所能表示的数的大小，即0xFF。我们在此声明了5 个字节，并将它们的值都初始化为0。

和指令不同，对于在程序中声明的数值，在编译阶段，编译器会在它们被声明的汇编地址处原样保留。

按照标准的做法，程序中用到的数据应当声明在一个独立的段，即数据段中。但是在这里，为方便起见，数据和指令代码是放在同一个段中的。不过，方便是方便了，但也带来了一个隐患，如果安排不当，处理器就有可能执行到那些非指令的数据上。尽管有些数碰巧和某些指令的机器码相同，也可以顺利执行，但毕竟不是我们想要的结果，违背了我们的初衷。

好在我们很小心，在本程序中把数据声明在所有指令之后，在这个地方，处理器的执行流程无法到达。

检测点5.2

找出下面代码片断中的错误。用nasmide 程序实际编译一下，看看结果如何。

data1 db 0x55,0xf000,0x0f

data2 dw 0x38,0x20,0x55aa

5.5.4　分解数的各个数位

源程序第41、42 行，是把代码段寄存器CS 的内容传送到通用寄存器CX，然后再从CX 传送到数据段寄存器DS。在此之后，数据段和代码段都指向同一个段。之所以这么做，是因为我们刚才声明的数据是和指令代码混在一起的，可以认为是位于代码段中。尽管在指令中访问这些数据可以使用段超越前缀“CS:”，但习惯上，通过数据段来访问它们更自然一些。

前面已经说过，要分解一个数的各个数位，需要做除法。8086 处理器提供了除法指令div，它可以做两种类型的除法。

第一种类型是用16 位的二进制数除以8 位的二进制数。在这种情况下，被除数必须在寄存器AX 中，必须事先传送到AX 寄存器里。除数可

以由8 位的通用寄存器或者内存单元提供。指令执行后，商在寄存器AL 中，余数在寄存器AH 中。比如：

```
div cl
div byte [0x0023]
```

前一条指令中，寄存器CL 用来提供8 位的除数。假如AX 中的内容是0x0005，CL 中的内容是0x02，指令执行后，CL 中的内容不变，AL 中的商是0x02，AH 中的余数是0x01。

后一条指令中，除数位于数据段内偏移地址为0x0023 的内存单元里。这条指令执行时，处理器将数据段寄存器DS 的内容左移4 位，加上偏移地址0x0023 以形成物理地址。然后，处理器再次访问内存，从那个物理地址处取得一个字节，作为除数同寄存器AX 做一次除法。

任何时候，只要是在指令中涉及内存地址的，都允许使用段超越前缀。比如：

```
div byte [cs:0x0023]
div byte [es:0x0023]
```

话又说回来了，在一个源程序中，通常不可能知道汇编地址的具体数值，只能使用标号。所以，指令中的地址部分更常见的形式是使用标号。比如：

```
dividnd dw 0x3f0
divisor db 0x3f

……

mov ax,[dividnd]
div byte [divisor]
```

上面的程序很有意思，首先，声明了标号dividnd 并初始化了一个字0x3f0 作为被除数；然后，又声明了标号divisor 并初始化一个字节0x3f 作为除数。

在后面的mov 和div 指令中，是用标号dividnd 和divisor 来代替被除数和除数的汇编地址。在编译阶段，编译器用具体的数值取代括号中的标号dividnd 和divisor。现在，假设dividnd 和divisor 所代表的汇编地址

分别是0xf000 和0xf002，那么，在编译阶段，编译器在生成这两条指令的机器码之前，会先将它们转换成以下的形式：

```
mov ax,[0xf000]
div byte [0xf002]
```

当第一条指令执行时，处理器用0xf000 作为偏移地址，去访问数据段（段地址在段寄存器DS 中），来取得内存中的一个字0x3F0，并把它传送到寄存器AX 中。

当第二条指令执行时，处理器采用同样的方法取得内存中的一个字节0x3F，用它来和寄存器AX 中的内容做除法。当然，除法指令div 的功能你是知道的。

说了这么多，其实是在强调标号和汇编地址的对应关系，以及如何在指令中使用符号化的偏移地址。

第二种类型是用32 位的二进制数除以16 位的二进制数。在这种情况下，因为16 位的处理器无法直接提供32 位的被除数，故要求**被除数的高16 位在DX 中，低16 位在AX 中**。

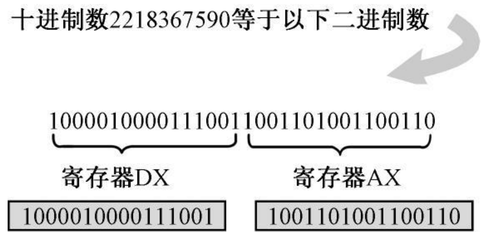


图5-6　用DX:AX 分解32 位二进制数示意图

这里有一个例子，如图5-6 所示，假如被除数是十进制数2218367590，那么，它对应着一个32 位的二进制数10000100001110011001101001100110。在做除法之前，先要分成两段进行“切割”，以分别装入寄存器DX 和AX 。为了方便， 我们通常用“DX:AX”来描述32 位的被除数。

同时，除数可以由16 位的通用寄存器或者内存单元提供，指令执行后，商在AX 中，余数在DX 中。比如下面的指令：

```
div cx
div word [0x0230]
```

源程序第45 行把0 传送到DX 寄存器，这意味着，我们是想把DX:AX 作为被除数，即被除数的高16 位是全零。至于被除数的低16 位，已经在第37 行的代码中被置为标号number 的汇编地址。

回到前面的第38 行，该指令把10 作为除数传送到通用寄存器BX 中。

一切都准备好了，源程序第46 行，div 指令用DX:AX 作为被除数，除以BX 的内容，执行后得到的商在AX 中，余数在DX 中。因为除数是10，余数自然比10 小，我们可以从DL 中取得。

第1 次相除得到的余数是个位上的数字，我们要将它保存到声明好的数据区中。所以，源程序第47 行，我们又一次用到了传送指令，把寄存器DL 中的余数传送到数据段。

可以看到，指令中没有使用段超越前缀，所以处理器在执行时，默认地使用段寄存器DS 来访问内存。偏移地址是由标号number 提供的，它是数据区的首地址，也可以说是数据区中第一个数据的地址。因此，number 和number+0x00 是一样的，没有区别。

因为我们访问的是number 所指向的内存单元，故要用中括号围起来，表明这是一个地址。

令人不解的是， 第47 行中， 偏移地址并非理论上的number+0x00 ， 而是0x7c00+ number+0x00。这个0x7c00 是从哪里来的呢？

标号number 所代表的汇编地址，其数值是在源程序编译阶段确定的，而且是相对于整个程序的开头，从0 开始计算的。请看一下表5-3 的第37 行，这个在编译阶段计算出来的值是0x012E。在运行的时候，如果该程序被加载到某个段内偏移地址为0 的地方，这不会有什么问题，因为它们是一致的。

但是，事实上，如图5-7 所示，这里显示的是整个0x0000 段，其中深色部分为主引导扇区所处的位置。主引导扇区代码是被加载到0x0000:0x7C00 处的，而非0x0000:0x0000。对于程序的执行来说，这不会有什么问题，因为主引导扇区的内容被加载到内存中并开始执行时，CS=0x0000， IP=0x7C00。

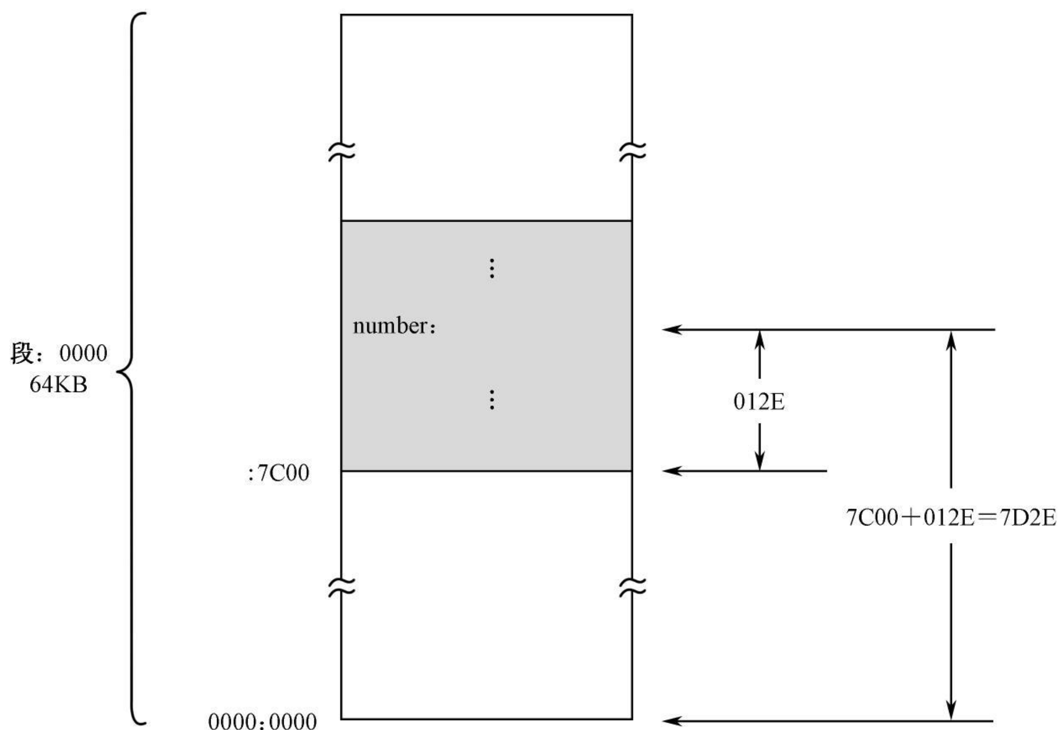


图5-7　主引导程序加载到内存后的地址变化

加载位置的改变不会对处理器执行指令造成任何困扰，但会给数据访问带来麻烦。要知道，当前

数据段寄存器DS 的内容是0x0000，因此，number 的偏移地址实际上是0x012E+0x7C00=0x7D2E。当正在执行的指令仍然用0x012E 来访问数据，灾难就发生了。

所以，在编写主引导扇区程序时，我们就要考虑到这一点，必须把代码写成

```
        mov [0x7c00+number+0x00],dl
```

指令中的目的操作数是在编译阶段确定的，因此，在编译阶段，编译器同样会首先将它转换成以下的形式，再进一步生成机器码：

```
        mov [0x7d2e],dl
```

这样，如表5-3 的第47 行所示，在编译后，编译器就会将这条指令编译成88 16 2E 7D，其中前两个字节是操作码，后两个字节是低端字节序的0x7D2E。当这条指令执行时，处理器将段寄存器DS 的内容（和CS

一样，是0x0000）左移4 位，再加上指令中提供的偏移地址0x7D2E，就得到了实际的物理地址（0x07D2E）。

关于这条指令的另外一个问题是，虽然目的操作数也是一个内存单元地址，但并没有用关键字“byte”来修饰。这是因为源操作数是寄存器DL，编译器可以据此推断这是一个字节操作，不存在歧义。

现在已经得到并保存了个位上的数字，下一步是计算十位上的数字，方法是用上一次得到的商作为被除数，继续除以10。恰好，AX 已经是被除数的低16 位，现在只需要把DX 的内容清零即可。

为此，代码清单5-1 第50 行，用了一个新的指令xor 来将DX 寄存器的内容清零。

xor，在数字逻辑里是异或（eXclusive OR）的意思，或者叫互斥或、互斥的或运算。《在穿越计算机的迷雾》里，已经花了大量的篇幅讲解数字逻辑。在数字逻辑里，如果0 代表假，1 代表真，那么

```
0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0
```

xor 指令的目的操作数可以是通用寄存器和内存单元，源操作数可以是通用寄存器、内存单元和立即数（不允许两个操作数同时为内存单元）。而且，异或操作是在两个操作数相对应的比特之间单独进行的。

一般地，xor 指令的两个操作数应当具有相同的数据宽度。因此，其指令格式可以总结为以下几种情况：

```
xor 8 位通用寄存器,8 位立即数, 例如: xor al,0x55
xor 8 位通用寄存器,指向 8 位实际操作数的内存地址, 例如: xor cl,[0x2000]
xor 8 位通用寄存器,8 位通用寄存器, 例如: xor bl,dl

xor 16 位通用寄存器,16 位立即数, 例如: xor ax,0xf033
xor 16 位通用寄存器, 指向 16 位实际操作数的内存地址, 例如: xor bx,[0x2002]
xor 16 位通用寄存器,16 位通用寄存器, 例如: xor dx,bx

xor 指向 8 位实际操作数的内存地址,8 位立即数, 例如: xor byte[0x3000],0xf0
xor 指向 8 位实际操作数的内存地址,8 位通用寄存器, 例如: xor [0x06],al

xor 指向 16 位实际操作数的内存地址,16 位立即数, 例如: xor word [0x2002],0x55aa
xor 指向 16 位实际操作数的内存地址,16 位通用寄存器, 例如: xor [0x20],dx
```

因为异或操作是在两个操作数相对应的比特之间单独进行，故，以下指令执行后，AX 寄存器中的内容为0xF0F3。

```
mov ax,0000_0000_0000_0010B
xor ax,1111_0000_1111_0001B        ;AX←1111_0000_1111_0011B, 即, 0xf0f3
```

注意，这两条指令的源操作数都采用了二进制数的写法，NASM 编译器允许使用下画线来分开它们，好处是可以更清楚地观察到那些感兴趣的比特。

回到当前程序中，因为指令xor dx,dx 中的目的操作数和源操作数相同，那么，不管DX 中的内容是什么，两个相同的数字异或，其结果必定为0，故这相当于将DX 清零。

值得一提的是，尽管都可以用于将寄存器清零，但是编译后，mov dx,0 的机器码是BA 00 00；而xor dx,dx 的机器码则是31 D2，不但较短，而且，因为xor dx,dx 的两个操作数都是通用寄存器，所以执行速度最快。

第二次相除的结果可以求得十位上的数字，源程序第52 行用来将十位上的数字保存到从number 开始的第2 个存储单元里，即number+0x01。

从源程序第55 行开始，一直到第67 行，做的都是和前面相同的事情，即，分解各位上的数字，并予以保存，这里不再赘述。

5.5.5 显示分解出来的各个数位

经过5 次除法操作，可以将寄存器AX 中的数分解成单独的数位，下面的任务是将这些数位显示出来，方法是从DS 指向的数据段依次取出这些数位，并写入ES 指向的附加段（显示缓冲区）。

因为在分解并保存各个数位的时候，顺序是“个、十、百、千、万”位，当在屏幕上显示时，却要反过来，先显示万位，再显示千位，等等，因为屏幕显示是从左往右进行的。所以，源程序第70 行，先从数据段中，偏移地址为number+0x04 处取得万位上的数字，传送到AL 寄存器。当然，因为程序是加载到0x0000:0x7C00 处的，所以正确的偏移地址是0x7C00+number+0x04。

然后，源程序第71 行，将AL 中的内容加上0x30，以得到与该数字对应的ASCII 代码。在这里，add 是加法指令，用于将一个数与另一个数相加。

add 指令需要两个操作数，目的操作数可以是8 位或者16 位的通用寄存器，或者指向8 位或者16 位实际操作数的内存地址；源操作数可以是相同数据宽度的8 位或者16 位通用寄存器、指向8 位或者16 位实际操作数的内存地址，或者立即数，但不允许两个操作数同时为内存单元。相加后，结果保存在目的操作数中。比如：

```
add al,cl                   ;寄存器 AL 和 CL 中的内容相加，结果在 AL 中
add ax,0x123f               ;寄存器 AX 的内容和立即数 0x123F 相加，结果在 AX 中
add [label_a],cx            ;内存单元的字和寄存器 CX 的内容相加，结果写回内存单元
add ax,[label_a]            ;寄存器 AX 的内容和内存单元的字相加，结果在 AX 中
add byte [label_a],0x08     ;内存单元的字节和立即数 0x08 相加，结果写回内存单元
```

源程序第72 行，将要显示的ASCII 代码传送到显示缓冲区偏移地址为0x1A 的位置，该位置紧接着前面的字符串“Label offset:”。显示缓冲区是由段寄存器ES 指向的，因此使用了段超越前缀。

源程序第73 行，将该字符的显示属性写入下一个内存位置0x1B。属性值0x04 的意思是黑底红字，无闪烁，无加亮。

从源程序的第75 行开始，到第93 行，用于显示其他4 个数位。

源程序第95、96 行，用于以黑底白字显示字符“D”，意思是所显示的数字是十进制的。

检测点5.3

1． INTEL x86 处理器访问内存时，是按低端字节序进行的。那么，以下程序片断执行后，寄存器AX 中的内容是多少？

```
    mov word [data],0x2008
    xor byte [data],0x05
    add word [data],0x0101
    mov ax,[data]
data db 0,0
```

2． 对于以上程序片断，如果标号data 在编译时的汇编地址是0x0030，那么，当该程序加载到内存后，该程序片断所在段的段地址为0x9020 时，该标号处的段内偏移地址和物理内存地址各是多少？

3． 对于以下指令的写法，说出哪些是正确的，哪些是错误的，错误的原因是什么。

A.mov ax,[data1]　　B.div [data1]　　C.xor ax,dx

D.div byte [data2]　　E.xor al,[data3]　　F.add [data4],0x05

G.xor 0xff,0x55　　H.add 0x06,al　　I.div 0xf0

J.add ax,cl

4． 如果寄存器AX、BX 和DX 的内容分别为0xA000、0x9000 和0x0001，那么，执行div bh 后，这三个寄存器的内容各是多少？执行div bx 后呢？

5.6　使程序进入无限循环状态

数字显示完成后，原则上整个程序就结束了，但对处理器来说，它并不知道。对它来说，取指令、执行是永无止境的。程序有大小，执行无停息，它这么做的结果，就是会执行到后面非指令的数据上，然后……

问题在于我们现在的确无事可做。为避免发生问题，源程序第98行，安排了一个无限循环：

```
infi: jmp near infi
```

jmp 是转移指令，用于使处理器脱离当前的执行序列，转移到指定的地方执行，关键字near表示目标位置依然在当前代码段内。上面这条指令唯一特殊的地方在于它不是转移到别处，而是转移到自己。也就是说，它将会不停地重复执行自己。不要觉得奇怪，这是允许的。

处理器取指令、执行指令是依赖于段寄存器CS 和指令指针寄存器IP 的，8086 处理器取指令时，把CS 的内容左移4 位，加上IP 的内容，形成20 位的物理地址，取得指令，然后执行，同时把IP 的内容加上当前指令的长度，以指向下一条指令的偏移地址。

但是，一旦处理器取到的是转移指令，情况就完全变了。

很容易想到，指令jmp near infi 的意图是转移到标号infi 所在的位置执行。可是，正如我们前面所说的，程序在内存中的加载位置是0x0000:0x7C00，所以，这条指令应当写成

```
jmp near 0x7c00+infi
```

实际上，不加还好，加上了0x7C00，就完全错了。

jmp 指令有多种格式。最典型地，它的操作数可以是直接给出的段地址和偏移地址，这称为绝对地址。比如：

```
jmp 0x5000:0xf0c0
```

此时，要转移到的目标位置是非常明确的，即，段地址为0x5000，段内偏移地址为0xf0c0。在这种情况下，指令的操作码为0xEA，故完整的机器指令是：

```
EA C0 F0 00 50
```

处理器执行时，发现操作码为0xEA，于是，将指令中给出的段地址传送到段寄存器CS；将偏移地址传送到指令指针寄存器IP，从而转移到目标位置处接着执行。

但是，在此处，jmp 指令使用了关键字“near”，且操作数是以标号（infi）的形式给出。这很容易让我们想到，这又是另一种形式的转移指令，转移的目标位置处在当前代码段内，指令中的操作数应当是目标位置的偏移地址。实际上，这是不正确的。

实际上，这是一个3 字节指令，操作码是0xE9，后跟一个16 位（两字节）的操作数。但是，该操作数并非目标位置的偏移地址，而是目标位置相对于当前指令处的偏移量（以字节为单位）。在编译阶段，编译器是这么做的：用标号（目标位置）处的汇编地址减去当前指令的汇编地址，再减去当前指令的长度（3），就得到了jmp near infi 指令的实际操作数。也不是编译器愿意费这个事，这是处理器的要求。这样看来，jmp near infi 的机器指令格式和它的汇编指令格式完全不同，颇具迷惑性，所以一定要认清它的本质。这种转移是相对的，操作数是一个相对量，如果你人为地加上0x7C00，那反而不对了。

那么，编译器是如何区分这两种不同的转移方式呢？很简单，当它看到JMP 之后是一个绝对地址，如0xF000:0x2000 时，它就知道应当编译成使用操作码0xEA 的直接绝对转移指令。相反地，如果它发现JMP 之后是一个标号，那么，它就会编译成使用操作码为0xE9 的相对转移指令。关键字“near”不是最主要的，它仅仅用于指示相对量是16 位的。

在这里，目标位置就是当前指令自己的位置，间隔的长度为0。再用0 减去当前指令长度3，这是一个负数。打开Windows 计算器程序，实际减一下看看，你会发现，用二进制的0 减去二进制的11，结果是

```
...... 1111111111111111111111111111111111111101
```

由于是在不断地向左边借位，除了最右边是01 外，左边都是无休止的“1”。

再切换到十六进制计算一下0x0 减去0x3，结果是

```
...... FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD
```

同样由于是在不断地向左边借位，除了最右边是D 外，左边都是无休止的F。

由于在指令中使用了near 关键字，因此，以上无休止的结果将被截断，只保留右边16 位，即0xFFFD。又因为x86 处理器使用低端字节序，所以，jmp near infi 指令编译后的机器代码为 E9 FD FF。

你可能觉得疑惑：0xFFFD 等于十进制数65533，而这条指令需要的操作数实际是负3，我们这样做的原理是什么呢？计算机又是怎么表示负数的呢？不要着急，下一章我们就要介绍负数，并回过头来重新认识这个问题。

在指令执行阶段，处理器用指令指针寄存器IP 的内容加上该指令的操作数，再加上该指令的长度（3），就得到了要转移的实际偏移地址，同时CS 寄存器的内容不变。因为改变了IP 的内容，这直接导致处理器的指令执行流程转向目标位置。

就jmp near infi 指令来说，当它执行时，转移到的目标位置是IP＋0xFFFD＋3。用Windows计算器程序实际做一下，0xFFFD＋3 的结果是0x10000，但处理器只使用16 位的偏移地址，故只保留16 位的结果0x0000。因此传送到指令指针寄存器IP 的内容依然是它原来的内容，这导致处理器再次执行当前指令。

jmp 指令具有多种格式，我们现在所用的，只是其中的一种，叫做相对近转移。有关其他格式，以及这些格式之间的差异，我们将在后面的章节里结合具体的实例进行讲解。

检测点5.4

写出以下程序片断中那两条JMP 指令的机器指令码，并在NASMIDE 中编译验证你的答案是否正确：

jmp near start （ ）

data db 0x55,0xaa

start: mov ax,0

jmp 0x2000:0x0005 （ ）

5.7 完成并编译主引导扇区代码

5.7.1 主引导扇区有效标志

主引导扇区在系统启动过程中扮演着承上启下的角色，但并非是唯一的选择。如果硬盘的主引导扇区不可用，系统还有其他选择，比如可以从光盘和U 盘启动。

然而，如果不试试水的深浅就一个猛子扎下池塘，这并非一个明智之举。同样地，如果主引导扇区是无效的，上面并非是一些处理器可以识别的指令，而处理器又不加鉴别地执行了它，其结果是陷入宕机状态，更不要提从其他设备启动了。

为此，计算机的设计者们决定，一个有效的主引导扇区，其最后两个字节的数据必须是0x55 和0xAA。否则，这个扇区里保存的就不是一些有意而为的数据。

定义这两个字节很简单，伪指令db 和dw 就可以实现。源程序第103 行就是db 版本的实现，但没有标号。标号的作用是提供当前位置的汇编（偏移）地址供其他指令引用，如果没有任何指令引用这个地址，标号可以省略。这是两个单独的字节，所以0x55 在前，0xAA 在后，即使编译之后也是这个顺序。

但是，如果采用dw 版本，应该这样写：

```
        dw 0xaa55
```

因为，在Intel 处理器上，将一个字写入内存时，是采用低端字节序的，低字节0x55 置入低地址端（在前），高字节0xAA 在高地址端（在后）。

麻烦在于，如何使这两个字节正好位于512 字节的最后。前面的代码有多少个字节我们不知道，那是由NASM 编译器计算和跟踪的。

我们当然有非常好的办法，但还不宜在这里说明。但是，经过计算和尝试，我知道，在前面的内容和结尾的0xAA55 之间，有203 字节的空洞。因此，源程序的第102 行，用于声明203 为0的数值来填补。

为了方便，伪指令times 可用于重复它后面的指令若干次。比如

```
times 20 mov ax,bx
```

将在编译时重复生成mov ax,bx 指令20 次，即重复该指令的机器码（89 D8）20 次。

因此

```
times 203 db 0
```

将会在编译时保留203 个为0 的字节。

5.7.2　代码的保存和编译

本章的代码是现成的，配书源代码解压缩之后，可以在文件夹“c05”里找到，文件名为c05_mbr.asm。打开该文件，将其编译成c05_mbr.bin。

该文件的大小为512 字节，可以用配书工具HexView 来查看其内容，如图5-8 所示。

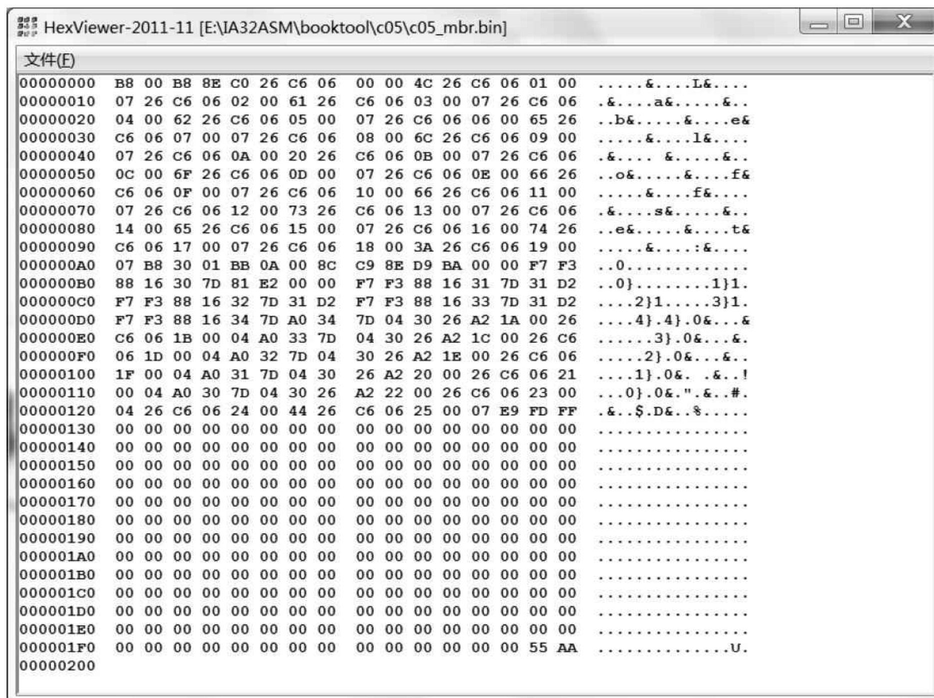


图5-8　用配书工具HexView 查看c05_mbr.bin 的内容

显而易见，在编译之后，源程序中的标号、注释、伪指令都统统消失了，只剩下纯粹的机器指令和数据。那些需要在编译阶段决定的内容，也都有了确切的值。

5.8 加载和运行主引导扇区代码

5.8.1 把编译后的指令写入主引导扇区

在第4 章，我们已经安装了VirtualBox 虚拟机软件，并在它里面创建了一台名为LEARNASM 的虚拟计算机。除此之外，还为它创建了一块虚拟硬盘。

虚拟硬盘其实是一个扩展名为".vhd"的Windows 文件，具体的文件名和创建位置只有你自己知道。但是，无论如何，你现在都可以将我们刚刚编译好的代码写入这个虚拟硬盘的主引导扇区里。

首先启动配书工具FixVhdWr，在第一个界面内选择虚拟硬盘文件。注意，要写入的那个虚拟硬盘，必须是VirtualBox 虚拟机使用的硬盘。否则的话，虚拟机怎么可能执行到你写入的程序呢！

接着，在第二个界面内，要选择刚才编译好的二进制文件c05_mbr.bin，然后再进入下一个界面。第三个界面开始将指定的文件写入虚拟硬盘。注意保持默认的写入方式（即"LBA 连续直写模式"），当出现红色字体的"数据写入完成，本次共操作了1 个扇区"时，说明数据的写入已经完成。最后要交待一句，千万不要在虚拟计算机LEARN-ASM 运行的时候进行数据写入操作，因为虚拟硬盘文件正被VirtualBox 以独占的方式使用。否则的话，会导致数据写入失败。

5.8.2 启动虚拟机观察运行结果

在Virtual Box 软件的主界面上，选择"LEARN-ASM"计算机，然后单击"运行"按钮。

最后，如果一切顺利的话，程序的运行效果如图5-9 所示。

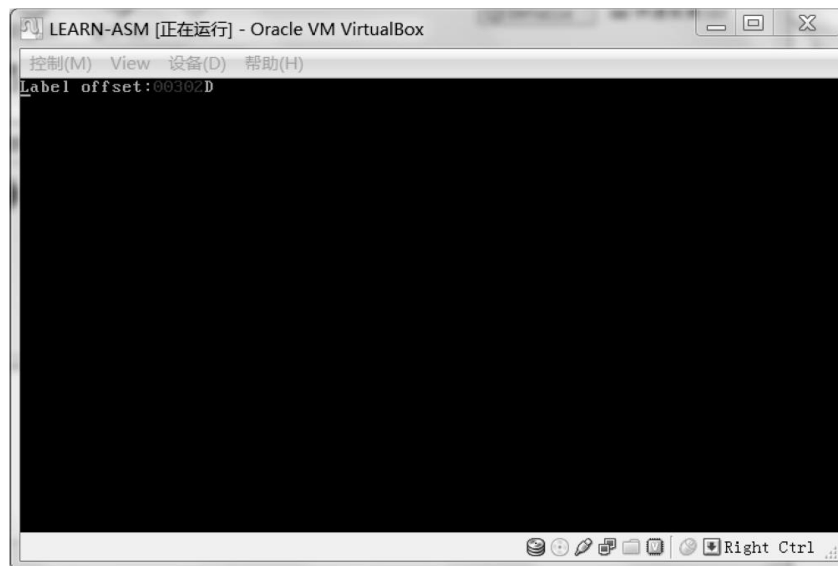


图5-9　本章程序在虚拟计算机中的运行效果

5.9 程序的调试技术

5.9.1 开源的Bochs 虚拟机软件

程序员的工作就像是在历险，困难重重，途中不可避免地要遇上暗礁。有时候，少了一个字符，或者多了一个字符，或者拼错了字符，程序就无法成功编译；有时候，尽管能够编译，但程序中存在逻辑错误，要么少写了语句，要么算法不对，运行的时候也得不到正确结果。

有时候，错误的原因很简单，就是因为马虎和误操作，但很难知道问题出在哪里。等到你终于发现的时候，一天，甚至几天的时间已经花掉了。在这种情况下，没有调试工具来找到程序中隐藏的错误是不行的。有时候，即使有调试工具的帮助，也会令人筋疲力尽，不过有总比没有好。在现实的世界里，不管是经验老道的程序设计师，还是刚入门的新手，没有谁敢说自己的程序是不需要调试的。

调试工具并不是智能到可以自动发现程序中的错误，这是不可能的。但是，它可以单步执行你的程序（每执行一条指令后就停下来），或者允许你在程序中设置断点，当它执行到断点位置时就停下来。这时，它可以显示处理器各个寄存器的内容，或者内存单元里的内容。因此，你可以根据机器的状态来判断程序的执行结果是否达到了预期。通过这种方式，你可以逐步逼近出现问题的地方，直到最终发现问题的所在。市面上有多种流行的程序调试工具软件，但它们通常都像你用的其他软件一样工作在操作系统之上。

麻烦的是，本书中的程序全都只能运行在没有操作系统的裸机下。这意味着，所有流行的调试工具都不可用。不过，好消息是，一款叫做 Bochs 的软件可以帮助你。

Bochs 是开源软件，是你唯一可选择的调试器。开源意味着，你不用花钱购买就可以使用它。它用软件来模拟处理器取指令和执行指令的过程，以及整个计算机硬件。当它开始运行时，就直接模拟计算机的加电启动过程。正是因为如此，它才有可能做一些调试工作。

很重要的一点是，它本身就是一个虚拟机，类似于VirtualBox。因此，它也就很容易让你单步跟踪硬盘的启动过程，查看寄存器的内容和

机器状态。在本书中，我们的程序都是直接从BIOS 那里接管处理器的控制权，因此，Bochs 的这个特点正好能够用来完成调试工作。不像本书中使用的其他工具，bochs 的使用方法在网上很容易搜索到。

要使用Bochs，首先要从它的官网下载安装程序。下载地址是：

```
http://sourceforge.net/projects/bochs/files/bochs/
```

在本书的配书文件包中，有一个关于如何下载、安装和配置Bochs的帮助文档，有WORD 和PDF 两种格式可以选用。请按照帮助文档的说明，安装和配置好Bochs。

一般来说，你会选择VirtualBox 虚拟机来观察运行结果，而在调试程序时使用Bochs。因此，最好是它们共用同一个虚拟硬盘文件（VHD 文件）。通过阅读帮助文档，你应该已经知道如何做到这一点，这里就不再赘述。

5.9.2 Bochs 下的程序调试入门

Bochs 虚拟机启动后，首先在当前的工作文件夹下寻找并读入配置文件bochsrc.bxrc，然后按它的参数调整当前虚拟机的各种“软硬件”配置和工作参数。

就像一台真正的计算机一样，Bochs 的“处理器”在加电之后，要开始取指令并执行指令。但是，与真正的处理器不同，如图5-10 所示，Bochs 在执行它启动之后的第一条指令时，会停下来，等待你的调试命令。

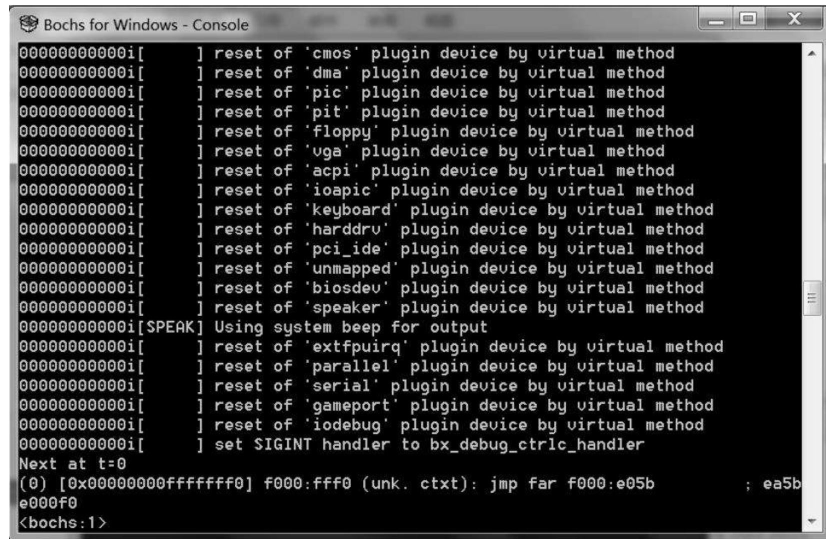


图5-10　Bochs 调试器的启动和断点命令的使用

如图中所示，命令窗口的底部显示了当前正在等待执行的那条指令，即“jmp far f000:e05b”。在这条指令中，关键字“far”是不必要的，而且在Bochs 中，数值默认是十六进制的。因此，该指令就是

```
jmp 0xf000:0xe05b
```

很显然，转移的目标位置是ROM-BIOS。

在那一行的左侧，显示了该指令所在的物理内存地址，该地址是用方括号围起来的。你可能会想，它怎么会是0x00000000FFFFFFF0 呢？

8086 有20 根地址线，加电启动之后，代码段寄存器（CS）的内容为0xFFFF，指令指针寄存器IP 的内容为0x0000，因此，第一条指令的物理地址是20 位的0xFFFF0。但是，8086 处理器已经成为历史，它之后的处理器都能够兼容8086 的功能，但却拥有超过32 根的地址线。在当前的这个Bochs 虚拟机上，地址线的数量是32 根。因此，Bochs 在这里用64 位的宽度来显示物理地址。但是，它的值应该是0x00000000000FFFF0，不是吗？

事情是这样的，和8086 不同，现代处理器在加电启动时，段寄存器CS 的内容为0xF000，指令指针寄存器IP 的内容为0xFFF0，这就使得处理器地址线的低20 位同样是0xFFFF0。这还不算完，在刚刚启动时，处理器将其余（高位部分）的地址线强制为高电平。因为当前Bochs 虚拟机的地址线是32 根，所以，初始发出的物理内存地址就是0x00000000FFFFFFF0 了。

之所以这样做，是因为处理器的设计者希望把ROM-BIOS 放到4GB（32 根地址线可提供的寻址范围是2^{32} ＝4GB）可寻址内存范围的最高端，这样，4GB 以下，连同传统的低端1MB 都是连续的RAM 区，连续的、不间断的RAM 能为操作系统管理内存带来方便。

问题在于，计算机制造商们会考虑很多现实问题。老的硬件和软件依赖于低端1MB 的ROMBIOS 来工作，这涉及到兼容性。最终，这两个地址区段都指向同一块ROM 芯片。

在物理地址的后边，是逻辑地址，即段寄存器CS 和指令指针寄存器IP 的内容，是以十六进制显示的，等效于0xf000:0xfff0。在这一行的右边，Bochs 还以注释的形式显示了指令的机器代码，即EA 5B E0 00 F0。

现在的情况是，Bochs 还没有执行该指令，它需要你的指示。此时，你可以单步执行指令。单步执行的意思是，每次只执行一条指令，执行完毕后再次停下来等待你的命令。

单步执行命令是“s”（step）。如图5-11 所示，输入“s”命令后回车，Bochs 执行刚才那条指令，然后停下来，同时显示下一条即将执行的指令。

```
Bochs for Windows - Console
00000000000i[     ] reset of 'floppy' plugin device by virtual method
00000000000i[     ] reset of 'vga' plugin device by virtual method
00000000000i[     ] reset of 'acpi' plugin device by virtual method
00000000000i[     ] reset of 'ioapic' plugin device by virtual method
00000000000i[     ] reset of 'keyboard' plugin device by virtual method
00000000000i[     ] reset of 'harddrv' plugin device by virtual method
00000000000i[     ] reset of 'pci_ide' plugin device by virtual method
00000000000i[     ] reset of 'unmapped' plugin device by virtual method
00000000000i[     ] reset of 'biosdev' plugin device by virtual method
00000000000i[     ] reset of 'speaker' plugin device by virtual method
00000000000i[SPEAK] Using system beep for output
00000000000i[     ] reset of 'extfpuirq' plugin device by virtual method
00000000000i[     ] reset of 'parallel' plugin device by virtual method
00000000000i[     ] reset of 'serial' plugin device by virtual method
00000000000i[     ] reset of 'gameport' plugin device by virtual method
00000000000i[     ] reset of 'iodebug' plugin device by virtual method
00000000000i[     ] set SIGINT handler to bx_debug_ctrlc_handler
Next at t=0
(0) [0x00000000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b        ; ea5b
e000f0
<bochs:1> s
Next at t=1
(0) [0x00000000000fe05b] f000:e05b (unk. ctxt): xor ax, ax               ; 31c0

<bochs:2> _
```

图5-11　在Bochs 中单步执行指令

如图中所示，指令执行后，下一条等待执行的指令为xor ax,ax，对应的机器指令码为31 C0，所在的物理内存地址是0x00000000000FE05B。注意，物理地址变了。

现代的x86 处理器在加电后，所有高端的地址线都被强制为高电平，直至遇到并执行了第一个段间转移指令。段间转移指令是在两个代码段之间实施控制转移，也就是同时改变段寄存器CS和指令指针寄存器IP 的JMP 指令，像jmp 0xf000:0xe05b 就是一个典型的例子。因此，当该指令执行后，处理器发出的物理地址就仅仅取决于CS 和IP 了。

接下来，你可以继续单步执行。但是，老在BIOS 中转悠也没什么意思。要知道，你调试的程序位于主引导扇区中。依靠单步执行，得什么时候才能执行到主引导扇区代码！

不用担心，Bochs 提供了断点指令“b”（break）。所谓断点，就是事先设置一个（物理）内存地址，当处理器执行到这个地址时，就自动停下来。因为计算机启动后，总是把主引导程序加载到物理内存地址0x7c00 处，所以，可以将这个地址设为断点。

如图5-12 所示，输入“b 0x7c00”。意思是，在处理器执行到地址0x7c00 处的那条指令时，就停下来。然后，再输入命令“c”。

```
Bochs for Windows - Console
00000000000i[     ] reset of 'vga' plugin device by virtual method
00000000000i[     ] reset of 'acpi' plugin device by virtual method
00000000000i[     ] reset of 'ioapic' plugin device by virtual method
00000000000i[     ] reset of 'keyboard' plugin device by virtual method
00000000000i[     ] reset of 'harddrv' plugin device by virtual method
00000000000i[     ] reset of 'pci_ide' plugin device by virtual method
00000000000i[     ] reset of 'unmapped' plugin device by virtual method
00000000000i[     ] reset of 'biosdev' plugin device by virtual method
00000000000i[     ] reset of 'speaker' plugin device by virtual method
00000000000i[SPEAK] Using system beep for output
00000000000i[     ] reset of 'extfpuirq' plugin device by virtual method
00000000000i[     ] reset of 'parallel' plugin device by virtual method
00000000000i[     ] reset of 'serial' plugin device by virtual method
00000000000i[     ] reset of 'gameport' plugin device by virtual method
00000000000i[     ] reset of 'iodebug' plugin device by virtual method
00000000000i[     ] set $IGINT handler to bx_debug_ctrlc_handler
Next at t=0
(0) [0x00000000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b        ; ea5b
e000f0
<bochs:1> s
Next at t=1
(0) [0x00000000000fe05b] f000:e05b (unk. ctxt): xor ax, ax               ; 31c0

<bochs:2> b 0x7c00
<bochs:3> c
```

图5-12　在Bochs 中设置断点

命令“c”（continue）是持续执行的意思，该命令要求处理器不间断地持续执行指令。但是，如果设置了断点，它就会在断点处停下来。因此，如图5-13 所示，当“c”命令执行后，它会在执行到物理内存地址0x7c00 时停下来。

```
Bochs for Windows - Console
00001213857i[CPU0 ] RSM: Resuming from System Management Mode
00001377875i[PCI  ] setting SMRAM control register to 0x0a
00001392768i[BIOS ] MP table addr=0x000fa510 MPC table addr=0x000fa440 size=0xc8

00001394524i[BIOS ] SMBIOS table addr=0x000fa520
00001394582i[MEM0 ] allocate_block: block=0x1f used 0x2 of 0x20
00001396719i[BIOS ] ACPI tables: RSDP addr=0x000fa640 ACPI DATA addr=0x01ff0000
size=0x992
00001399916i[BIOS ] Firmware waking vector 0x1ff00cc
00001401337i[PCI  ] 440FX PMC write to PAM register 59 (TLB Flush)
00001402065i[BIOS ] bios_table_cur_addr: 0x000fa664
00001529682i[VBIOS] VGABios $Id: vgabios.c,v 1.75 2011/10/15 14:07:21 vruppert E
xp $
00001529753i[BXVGA] VBE known Display Interface b0c0
00001529785i[BXVGA] VBE known Display Interface b0c5
00001532710i[VBIOS] VBE Bios $Id: vbe.c,v 1.64 2011/07/19 18:25:05 vruppert Exp
$
00001872474i[BIOS ] ata0-0: PCHS=1003/12/17 translation=none LCHS=1003/12/17
00005749680i[BIOS ] IDE time out
00017824978i[BIOS ] Booting from 0000:7c00
(0) Breakpoint 1, 0x0000000000007c00 in ?? ()
Next at t=17825033
(0) [0x0000000000007c00] 0000:7c00 (unk. ctxt): mov ax, 0xb800           ; b800
b8
<bochs:4>
```

图5-13　Bochs 执行到主引导扇区代码时的状态

如图中所示，当前等待执行的指令是mov ax,0xb800，这就是本章源代码的第一条指令；该指令的物理地址是0x0000000000007C00，指令的机器代码为B8 00 B8。

如图5-14 所示，此时，可以输入命令“r”（register）来显示通用寄存器的内容。

```
Bochs for Windows - Console
00001529682i[VBIOS] VGABios $Id: vgabios.c,v 1.75 2011/10/15 14:07:21 vruppert E
xp $
00001529753i[BXVGA] VBE known Display Interface b0c0
00001529785i[BXVGA] VBE known Display Interface b0c5
00001532710i[VBIOS] VBE Bios $Id: vbe.c,v 1.64 2011/07/19 18:25:05 vruppert Exp
$
00001872474i[BIOS ] ata0-0: PCHS=1003/12/17 translation=none LCHS=1003/12/17
00005749680i[BIOS ] IDE time out
00017824978i[BIOS ] Booting from 0000:7c00
(0) Breakpoint 1, 0x0000000000007c00 in ?? ()
Next at t=17825033
(0) [0x0000000000007c00] 0000:7c00 (unk. ctxt): mov ax, 0xb800           ; b800
b8
<bochs:4> r
rax: 0x00000000_0000aa55 rcx: 0x00000000_00090000
rdx: 0x00000000_00000080 rbx: 0x00000000_00000000
rsp: 0x00000000_0000ffd6 rbp: 0x00000000_00000000
rsi: 0x00000000_000e0000 rdi: 0x00000000_0000ffac
r8 : 0x00000000_00000000 r9 : 0x00000000_00000000
r10: 0x00000000_00000000 r11: 0x00000000_00000000
r12: 0x00000000_00000000 r13: 0x00000000_00000000
r14: 0x00000000_00000000 r15: 0x00000000_00000000
rip: 0x00000000_00007c00
eflags 0x00000082: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af pf cf
<bochs:5>
```

图5-14　用“r”命令显示通用寄存器的内容

我知道，对于图中的内容，你一定会摇摇头表示看不懂，这其实很正常。我们此时正在介绍8086 处理器，如图5-15 所示，它有8 个16 位的通用寄存器AX、BX、CX、DX、SI、DI、BP 和SP。其中，前4 个寄存器还可以各自分成两个独立的8 位寄存器来用，即AH、AL、BH、BL、CH、CL、DH 和DL；后4 个寄存器只能作为16 位寄存器整体使用。除此之外，从图中可以看出，它的指令指针寄存器IP 也是16 位的。

正如你已经知道的，8086 已经成为历史，现在我们所使用的处理器，都是32 位或者64 位的。32 位x86 处理器对寄存器做了扩展，使之达到32 位，以处理32 位的数据。如图中所示，这8 个32 位寄存器分别是EAX、EBX、ECX、EDX、ESI、EDI、EBP 和ESP，它们可以在程序中直接做为32 位寄存器使用。同时，指令指针寄存器IP 也做了扩展，达到32 位，即EIP。为了保持同8086 的兼容性，这些寄存器的低16 位依然保持以前的用法，这使得以前的程序可以在32 位处理器上正常运行。

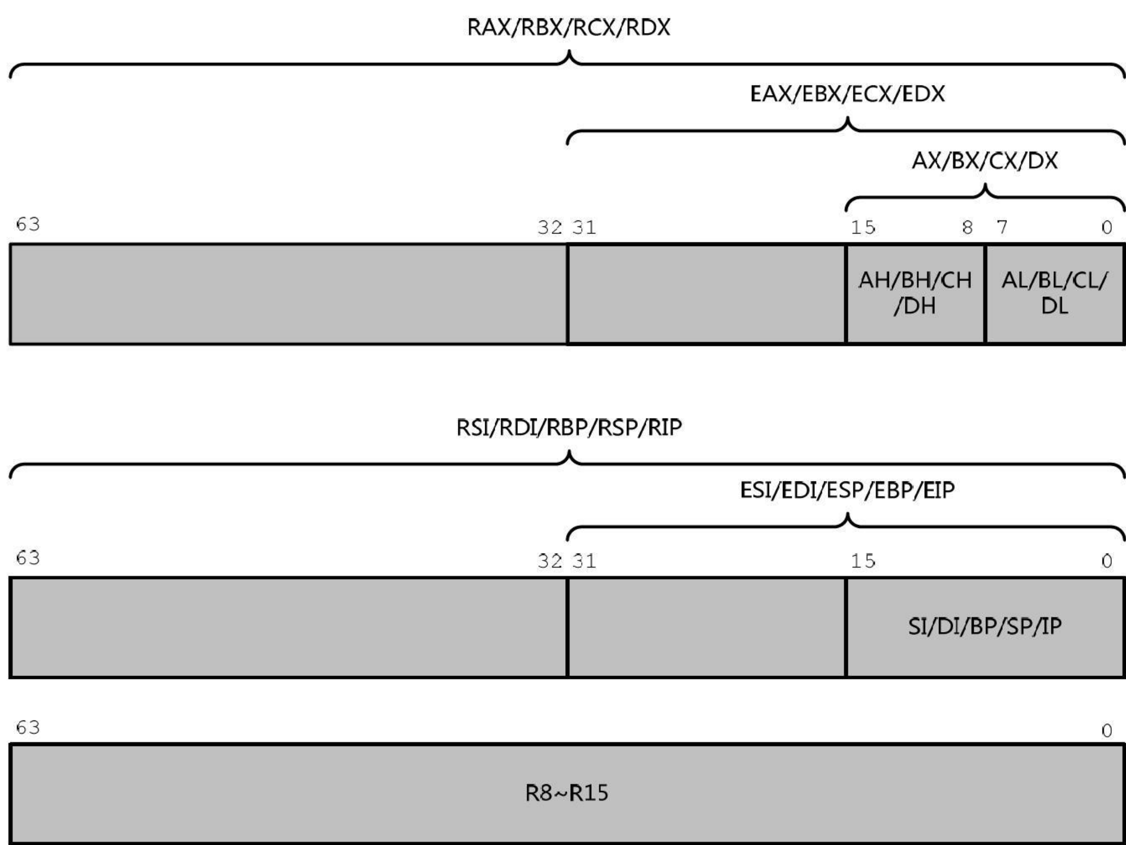


图5-15　通用寄存器的扩展示意图

在64 位处理器上，这些寄存器再次被扩展，达到了64 位，即RAX、RBX、RCX、RDX、RSI、RDI、RBP、RSP 和RIP。同时，它们的低32 位（包括低16 位）依然保持从前的用法。

除此之外，64 位的x86 处理器还新增了8 个64 位的寄存器R8、R9、R10、R11、R12、R13、R14 和R15，它们只能整体作为64 位的寄存器来用。

屏幕的底部还显示了标志寄存器EFLAGS 的状态。有关标志寄存器的内容将在后面的章节里具体阐述，这里先不用管它。有关32 位处理器的内容，将在本书的后半部分讲解；有关64 位处理器的内容，将在本套图书的其他分册讲解。

注意，尽管Bochs 把所有寄存器都显示为64 位的宽度，如RAX，但这并不表明你的处理器就一定是64 位的。它的目的很简单，仅仅是希望用同一种最宽的格式来应付所有不同的处理器。

这样一说你就应该很清楚了，如前图5-14 所示，RAX 的内容是0x000000000000aa55，这就意味着，RAX 的高48 位是全零，低16 位（即AX）是0xAA55。

再比如那幅图中，RIP 的内容是0x0000000000007c00，它表明RIP 寄存器的高48 位是全零，低16 位（即IP）是0x7C00。

我们调试到哪一步了？

如图5-14 所示，当前正在等待执行指令是mov ax,0xb800。现在，我们用“s”命令单步执行该指令。如图5-16 所示，单步执行之后，下一条等待执行的指令是mov es,ax，该指令的物理内存地址是0x0000000000007C03。

因为刚才那条指令是将立即数0xB800 传送到寄存器AX，那么，我们现在可以用“r”命令来看看寄存器AX 的内容是否真的发生了改变。如图中所示，寄存器AX 的内容是0xB800，确实符合我们的预期。

```
Bochs for Windows - Console
rdx: 0x00000000_00000080 rbx: 0x00000000_00000000
rsp: 0x00000000_0000ffd6 rbp: 0x00000000_00000000
rsi: 0x00000000_000e0000 rdi: 0x00000000_0000ffac
r8 : 0x00000000_00000000 r9 : 0x00000000_00000000
r10: 0x00000000_00000000 r11: 0x00000000_00000000
r12: 0x00000000_00000000 r13: 0x00000000_00000000
r14: 0x00000000_00000000 r15: 0x00000000_00000000
rip: 0x00000000_00007c00
eflags 0x00000082: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af pf cf
<bochs:5> s
Next at t=17825034
(0) [0x0000000000007c03] 0000:7c03 (unk. ctxt): mov es, ax             ; 8ec0

<bochs:6> r
rax: 0x00000000_0000b800 rcx: 0x00000000_00090000
rdx: 0x00000000_00000080 rbx: 0x00000000_00000000
rsp: 0x00000000_0000ffd6 rbp: 0x00000000_00000000
rsi: 0x00000000_000e0000 rdi: 0x00000000_0000ffac
r8 : 0x00000000_00000000 r9 : 0x00000000_00000000
r10: 0x00000000_00000000 r11: 0x00000000_00000000
r12: 0x00000000_00000000 r13: 0x00000000_00000000
r14: 0x00000000_00000000 r15: 0x00000000_00000000
rip: 0x00000000_00007c03
eflags 0x00000082: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af pf cf
<bochs:7>
```

图5-16　观察指令执行后的效果（寄存器AX 的变化）

接下来，继续用单步指令“s”来执行mov es,ax 指令。如图5-17 所示，该指令执行后，下一条即将执行的指令是

```
mov byte ptr es:0x0,0x4c
```

Bochs 的汇编指令格式和NASM 相比，在某些方面是不同的。实际上，这条指令就是本章程序中的

```
mov byte [es:0x00],'L'
```

因为字面值’L’早在程序编译时就被转换成了立即数0x4c，所以，严格地说，这条指令在NASM 中的格式是

```
mov byte [es:0x0],0x4c
```

无论如何，这条指令还没有执行，刚才执行的是mov es,ax 指令。此时，段寄存器ES 中的内容应当是0xB800。

为了验证这一点，应当要求Bochs 显示段寄存器的内容。为此，需要使用“sreg”（segment register）命令。如图中所示，当输入“sreg”命令后，Bochs 显示了一大堆东西。

在32 位和64 位处理器中，除了段寄存器CS、SS、DS 和ES 外，还新增了两个段寄存器FS 和GS，这一点首先要明白。

然后，在32 位和64 位处理器中，以上6 个段寄存器都依然是16 位的，但都额外增加了一个不可访问的部分，叫做段描述符高速缓存器。段描述符高速缓存器由处理器内部使用，不能在程序中访问，里面存放了段的起始地址、段的扩展范围，以及段和各种属性，比如它是代码段还是数据段，是否可以写入，是否被访问过，等等。这些知识，将在本书的后半部分详细讲解。

如图中所示，Bochs 首先显示了段寄存器ES 中的内容，是0xb800，这符合我们的预期。同时，它还显示了ES 描述符高速缓存器的内容，因为还没有讲到，所以暂时不用管它。

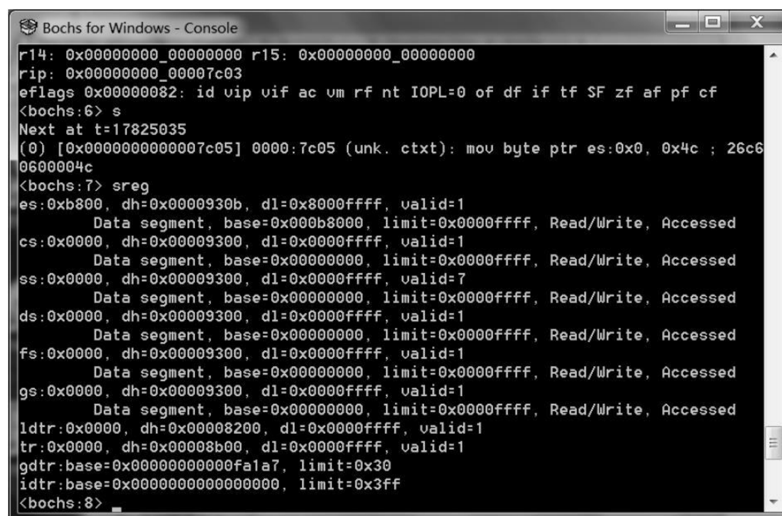


图5-17　在Bochs 中显示段寄存器的内容

接下来，如图5-18 所示，我们连续单步执行两次。对照本章的源程序，这实际上是执行了以下两条指令：

```
mov byte [es:0x00],'L'
mov byte [es:0x01],0x07
```

我们知道，这是在写文本模式下的显示缓冲区。因此，从物理内存地址0xB8000 处开始的两个字节必然是0x4C 和0x07。

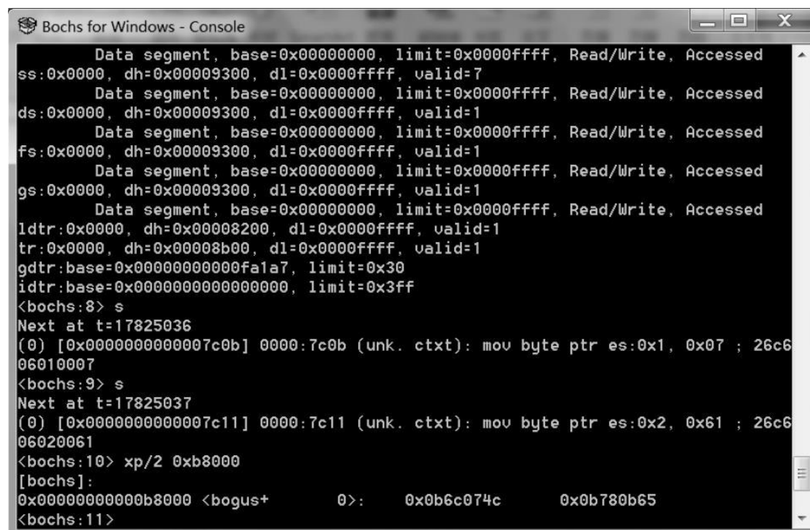


图5-18　显示内存区域中的内容

为了验证这一点，需要显示内存中的内容，这可以使用命令“xp”（eXamine memory at Physical address），即，显示指定物理内存地址

处的内容。xp 命令每次只显示一个双字。要显示多个双字，需要用“/”附加一个数量。然后，还应当指定一个物理内存地址。

如图5-18 所示，在这里，我们要求从物理内存地址0xB8000 开始，显示2 个双字。很快，Bochs 做出了回应，显示了两个双字0x0b6c074c 和0x0b780b65。

如图5-19 所示，双字数据在内存中的存放是按低端字节序的。因此，0x0b6c074c 这个双字数据，在内存中对应着从物理地址0xB8000 开始的4 个字节0x4C、0x07、0x6C 和0x0B。

至此，基本的程序调试技术就讲完了，你可以使用“q”（quit）命令退出Bochs 调试过程。

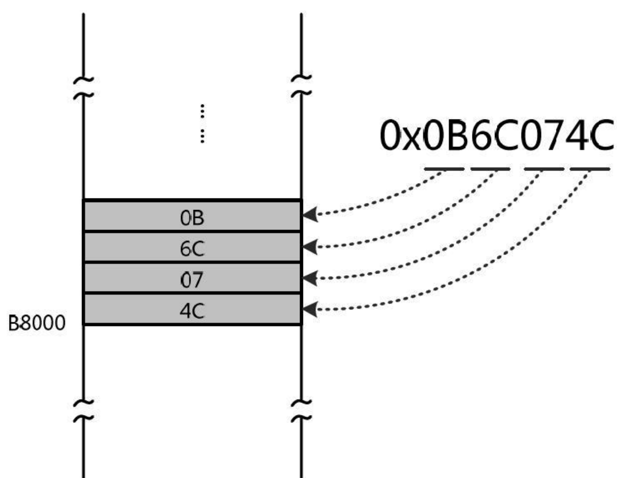


图5-19　以低端字节序分析双字在内存中的位置

检测点5.5

1．在你自己的计算机上重现以上的编译、运行（使用VirtualBox）和调试（使用bochsdbg）过程。

2．单步执行本章程序，观察div 指令执行后的寄存器内容变化。

本章习题

1．试找出以下程序片断中隐藏的问题并进行修正：

```
mov ax,21015
mov bl,10
div bl
and cl,0xf0
```

2．本章的程序在内存中的加载地址是0x0000:0x7C00，此时，指令jmp near infi 在段内的偏移地址是多少？试修改本章的源程序以显示该值。

3．汇编语言编译器采用助记符来方便指令的书写和阅读。比如，mov 是传送指令，div 是除法指令。假如Intel 公司新推出一款处理器，该处理器新增了一条指令，其机器码为CD 88。因为是新指令，你的NASM 编译器肯定没有一个助记符与之相对应。在这种情况下，如何在你的程序中使用该指令？

第6章　相同的功能，不同的代码

汇编语言是最有效率的计算机语言，由于直接面向处理器编程，编译后的机器代码执行起来速度也是最快的。为了进一步讲解汇编语言的指令和语法，在本章里，我们采用不同的方法来实现和上一章相同的功能。本章的学习目标是：

1．用一种不同的分段方法，从另一个不同的角度理解处理器的分段内存访问机制。

2．在计算机中，指令的执行并非总是按照它们的自然排列顺序来进行的，其执行流程也会因为各种原因发生变化。本章将学习两种非顺序的程序流程控制方法，即循环和条件转移。

3．认识几种新指令，包括movsb、movsw、inc、dec、cld、std、div、neg、cbw、cwd、sub、idiv、jcxz、cmp 等。

4．认识INTEL8086 标志寄存器FLAGS 的各个标志位，了解条件转移指令。

5．认识计算机中的负数。

6．学习用Bochs 调试程序的更多技巧，包括察看FLAGS 寄存器各标志位的状态。

6.1　代码清单6-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：6-1（主引导扇区程序）

源程序文件：c06_mbr.asm

6.2 跳过非指令的数据区

如代码清单6-1 所示，从源程序第8 行到第10 行，声明了非指令的数据。一般来说，所有处理器指令都应当顺序存放，在它们中间不允许夹杂非指令的普通数据，因为它们不能作为指令执行。但是，如果有办法让处理器执行不到这些非指令的内容，则又另当别论。为此，在这些数据之前，源程序的第6 行，是一条转移指令

```
        jmp near start
```

在这里，该指令用来使处理器的执行流越过这些不可执行的数据，转移到后面标号start 处的代码接着执行。

正如我们在上一章里讲到的，像jmp near start 这种指令，机器指令的操作码是0xE9，操作数是一个16 位的相对偏移量，这叫做相对近转移，后面我们还要继续讨论这个话题。

6.3 在数据声明中使用字面值

在第5 章中，显示字符串“Label offset:”的方法是将每个字符的ASCII码包含在每条指令中，即它们是作为每条指令的操作数出现的。这种方法很原始，也很笨拙。而且，如果要改变显示的内容，则必须重新编写指令，很不方便。

在本章中，我们将要改变这种做法，使得显示字符串的手段更灵活，具体做法是专门定义一个存放字符串的数据区，当要显示它们的时候，再用指令取出来，一个一个地传送到显示缓冲区。这样一来，负责在屏幕上显示的指令就和要显示的内容无关了。

源程序的第8、9 行，这两行的目的是声明要显示的内容。在NASM里，“\”是续行符，当一行写不下时，可以在行尾使用这个符号，以表明下一行与当前行应该合并为一行。

和上一章相同，在用伪指令db 声明字符的ASCII 码数据时也可以使用字面值。在编译阶段，编译器将把’L’、’a’等转换成与它们等价的ASCII代码。

除了ASCII 码，这里还声明了每个字符的显示属性值0x07，都是已经讲过的知识，相信很好理解。

6.4 段地址的初始化

汇编语言源程序的编译符合一种假设，即编译后的代码将从某个内存段中，偏移地址为0 的地方开始加载。这样一来，如果有一个标号“label_a”，它在编译时计算的汇编地址是0x05，那么，当程序被加载到内存后，它在段内的偏移地址仍然是0x05，任何使用这个标号来访问内存的指令都不会产生问题。

但是，如果程序加载时，不是从段内偏移地址为0 的地方开始的，而是0x7c00，那么，label_a 的实际偏移地址就是0x7c05。这时，所有访问label_a 的指令仍然会访问偏移地址0x05，因为这是在编译时就决定了的。实际上，这样的问题在上一章就遇到过。在那里，因为我们已经知道程序将来的加载位置是0x0000:0x7c00，所以才有了这样古怪的写法：

```
        mov [0x7c00+number+0x00],dl
```

不得不说，0x7c00 就是理论和现实之间的差距。

在主引导程序中，访问内存的指令很多，如果都要加上0x7c00 无疑是很麻烦的，这个我们已经看到了。其实，产生这个问题的根源，就是因为程序在加载时，没有从段内偏移地址为0 的地方开始。

好在 Intel 处理器的分段策略还是很灵活的，逻辑地址0x0000:0x7c00 对应的物理地址是0x07c00，该地址又是段0x07C0 的起始地址。因此，这个物理地址其实还对应着另一个逻辑地址0x07c0:0000，如图6-1 所示。

看到了吧？我们可以把这512 字节的区域看成一个单独的段，段的基地址是0x07C0，段长512 字节。注意，该段的最大长度可以为64KB，但是在这里，我们实际上仅使用512 个字节。尽管BIOS 将主引导扇区加载到物理地址0x07c00 处，但我们却可以认为它是从0x07c0:0x0000 处开始加载的。

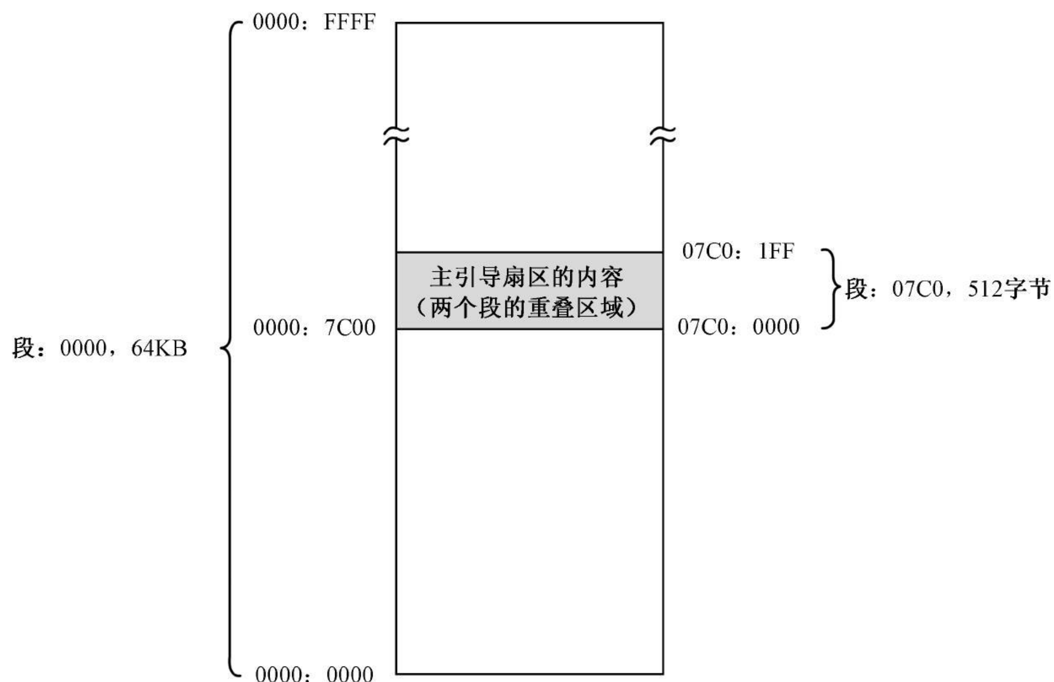


图6-1　以两个逻辑段的视角看待同一个内存区域

在这种情况下，如果执行指令

```
mov [0x05],dl
```

那么，处理器将把数据段寄存器DS 的内容（0x07c0）左移4 位，加上指令中指定的偏移地址（0x05），形成物理内存地址0x07c05，并将寄存器DL 中的内容传送到该处。

所以，源程序第13、14 行，通过传送指令将数据段寄存器DS 的内容设置为0x07c0。和以前一样，源程序第16、17 行，使附加段寄存器ES 的内容指向显示缓冲区所在的段0xb800。

6.5 段之间的批量数据传送

在本章中，要在屏幕上显示的内容，连同它们的显示属性值，都集中声明在一起。想显示它们？那就要将它们“搬”到0xB800 段。有多种方法可以做到这一点，但8086 处理器提供了最好的方法，那就是使用movsb 或者movsw 指令。

这两个指令通常用于把数据从内存中的一个地方批量地传送（复制）到另一个地方，处理器把它们看成是数据串。但是，movsb 的传送是以字节为单位的，而movsw 的传送是以字为单位的。

movsb 和movsw 指令执行时，原始数据串的段地址由DS 指定，偏移地址由SI 指定，简写为DS:SI；要传送到的目的地址由ES:DI 指定；传送的字节数（movsb）或者字数（movsw）由CX指定。除此之外，还要指定是正向传送还是反向传送，正向传送是指传送操作的方向是从内存区域的低地址端到高地址端；反向传送则正好相反。正向传送时，每传送一个字节（movsb）或者一个字（movsw），SI 和DI 加1 或者加2；反向传送时，每传送一个字节（movsb）或者一个字（movsw）时，SI 和DI 减去1 或者减去2。不管是正向传送还是反向传送，也不管每次传送的是字节还是字，每传送一次，CX 的内容自动减一。

如图6-2 所示，在8086 处理器里，有一个特殊的寄存器，叫做标志寄存器FLAGS。作为一个例子，它的第6 位是ZF（Zero Flag），即零标志。当处理器执行一条算术或者逻辑运算指令后，算术逻辑部件送出的结果除了送到指令中指定位置（目的操作数指定的位置）外，还送到一个或非门。学过逻辑电路课程，或者看过《穿越计算机的迷雾》这本书的人都知道，或非门的输入全为0 时，输出为1；输入不全为0，或者全部为1 时，输出为0。或非门的输出送到一个触发器，这就是标志寄存器的ZF 位。这就是说，如果计算结果为0，这一位被置成1，表示计算结果为零是“真”的；否则清除此位（0）。

除此之外，它也允许通过指令设置一些标志，来改变处理器的运行状态。比如，第10 位是方向标志DF（Direction Flag），通过将这一位清零或者置1，就能控制movsb 和movsw 的传送方向。

源程序第19 行是方向标志清零指令cld。这是个无操作数指令，与其相反的是置方向标志指令std。cld 指令将DF 标志清零，以指示传送是正方向的。和cld 功能相反的是std 指令，它将DF标志置位（1）。此时，传送的方向是从高地址到低地址。

源程序第20 行，设置SI 寄存器的内容到源串的首地址，也就是标号mytext 处的汇编地址。

源程序第21 行，设置目的地的首地址到DI 寄存器。屏幕上第一个字符的位置对应着0xB800段的开始处，所以设置DI 的内容为0。

第22 行，设置要批量传送的字节数到CX 寄存器。因为数据串是在两个标号number 和mytext 之间声明的，而且标号代表的是汇编地址，所以，汇编语言允许将它们相减并除以2 来得到这个数值。需要说明的是，这个计算过程是在编译阶段进行的，而不是在指令执行的时候。除以2 的原因是每个要显示的字符实际上占两字节：ASCII 码和属性，而movsw 每次传送一个字。

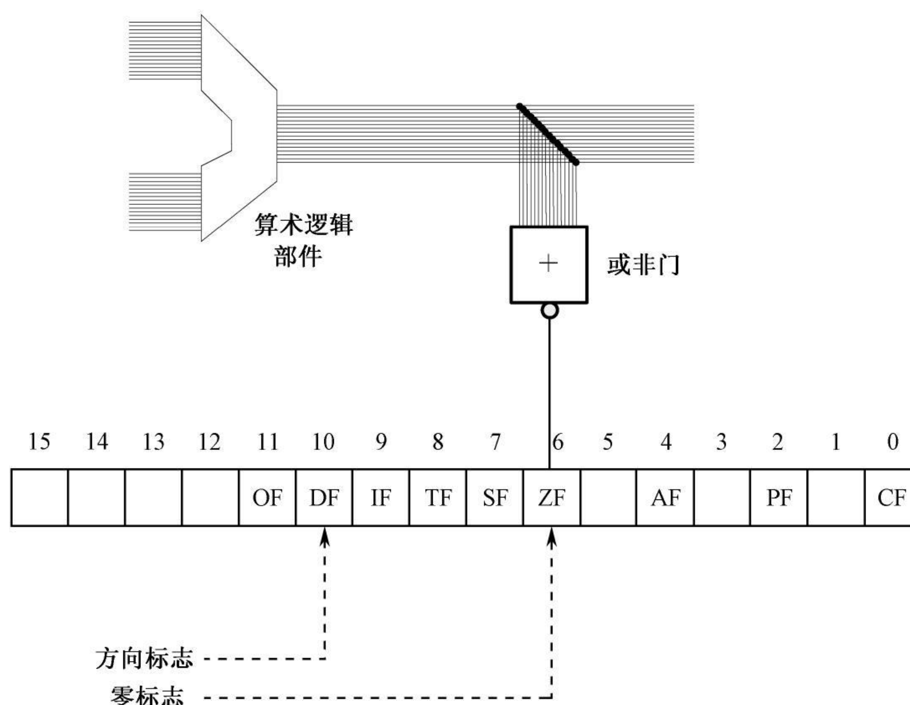


图6-2　8086 处理器的标志寄存器

第23 行，是movsw 指令，操作码是0xA5，该指令没有操作数。使用movsw 而不是movsb的原因是每次需要传送一个字（ASCII 码和属性）。单纯的movsb 和movsw 只能执行一次，如果希望处理器自动地反

复执行，需要加上指令前缀rep（repeat），意思是CX 不为零则重复。rep movsw 的操作码是0xF3 0xA5，它将重复执行movsw 直到CX 的内容为零。

检测点6.1

选择填空：MOVSW 指令每次传送一个（ ），MOVSW 指令每次传送一个（ ）。原始数据在段内的偏移地址在寄存器（ ）中，要传送的目标位置的偏移地址在寄存器（ ）中。如果要连续传送多个字或字节，则需要（ ）前缀，在寄存器（ ）中设置传送的次数，并设置传送的方向。其中，（ ）指令指示正向传送，（ ）指令指示反向传送。反向传送时，每传送一次，SI 和DI 的内容将（ ）。

A.字节　B.字　C.di　D.si　E.cx　F.rep　G.减小　H.std　I.cld　J.增大

6.6 使用循环分解数位

为了显示标号number 所代表的汇编地址，源程序第26 行用于将它的数值传送到寄存器AX，这个和以前是一样的。

声明标号number 并从此处开始初始化5 字节的目的主要是保存数位，但同时我们还想显示它的汇编地址。为了访问标号number 处的数位，需要获取它在内存段中的偏移地址。

为此，源程序第29 行，通过将AX 的内容传送到BX，来使BX 指向该处的偏移地址。实际上，这等效于

```
mov bx,number
```

只不过用寄存器传递来得更快，更方便。

第29～37 行依旧做的是分解数位的事，但用了和以往不同的方法。简单地说，就是循环。循环依靠的是循环指令loop，该指令出现在源程序的第37 行：

```
loop digit
```

loop 指令的功能是重复执行一段相同的代码，处理器在执行它的时候会顺序做两件事：

```
将寄存器 CX 的内容减一；
如果 CX 的内容不为零，转移到指定的位置处执行，否则顺序执行后面的指令。
```

和源程序第6 行的jmp near start 一样，loop digit 指令也是颇具迷惑性的指令，它的机器指令操作码是0xE2，后面跟着一个字节的操作数，而且也是相对于标号处的偏移量，是在编译阶段，编译器用标号digit 所在位置的汇编地址减去loop 指令的汇编地址，再减去loop 指令的长度（2）来得到的。

为了使loop 指令能正常工作，需要一些准备。源程序第30 行，将循环次数传送到CX 寄存器。因为分解AX 中的数需要循环5 次，故传送的值是5。

源程序第31 行，将除数10 传送到寄存器SI。

源程序第33～37 行是循环体，每次循环都会执行这些代码，主要是做除法并保存每次得到的余数。每次除法之前都要先将DX 清零以得到被除数的高16 位，这是源程序第33 行所做的事情。

做完除法之后，第35 行，将DL 中得到的余数传送到由BX 所指示的内存单元中去。这是我们第一次接触到偏移地址来自于寄存器的情况，而在此之前，我们仅仅是使用类似于下面的指令：

```
mov [0x05],dl
mov [number],al
mov [number+0x02],cl
```

尽管方式不同，但mov [bx],dl 做相同的事情，那就是把DL 中的内容，传送到以DS 的内容为段地址，以BX 的内容为偏移地址的内存单元中去。注意，指令中的中括号是必需的，否则就是传送到BX 中，而不是BX 的内容所指示的内存单元了。

在8086 处理器上，如果要用寄存器来提供偏移地址，只能使用BX、SI、DI、BP，不能使用其他寄存器。所以，以下指令都是非法的：

```
mov [ax],dl
mov [dx],bx
```

原因很简单，寄存器BX 最初的功能之一就是用来提供数据访问的基地址，所以又叫基址寄存器（Base Address Register）。之所以不能用SP、IP、AX、CX、DX，这是一种硬性规定，说不上有什么特别的理由。而且，在设计8086 处理器时，每个寄存器都有自己的特殊用途，比如AX是累加器（Accumulator），与它有关的指令还会做指令长度上的优化（较短）；CX 是计数器（Counter）；DX 是数据（Data）寄存器，除了作为通用寄存器使用外，还专门用于和外设之间进行数据传送；SI 是源索引寄存器（Source Index）；DI 是目标索引寄存器（Destination Index），用于数据传送操作，我们已经在movsb 和movsw 指令的用法中领略过了。

注意，可以在任何带有内存操作数的指令中使用BX、SI 或者DI 提供偏移地址。

做完一次除法，并保存了数位之后，源程序第36 行，用于将BX 中的内容加一，以指向下一个内存单元。inc 是加一指令，操作数可以是8 位或者16 位的寄存器，也可以是字节或者字内存单元。从功能上讲，它和

```
add bx,1
```

是一样的，但前者的机器码更短，速度更快。下面是两个例子：

```
inc al
inc byte [bx]
inc word [label_a]
```

以上，第一条指令执行时，处理器将寄存器AL 中的内容加一；第二条指令执行时，将寄存器BX 所指向的内存单元的内容加一。就是说，处理器用段寄存器DS 的内容左移4 位，加上寄存器BX 的内容，形成20 位物理地址。然后，将该地址处的内容（字节）加一。

第三条指令做和第二条指令相同的事情，但是偏移地址是用标号给出的。关键字“word”表明它操作的是内存中的一个字，段地址在段寄存器DS 中，偏移地址等于标号label_a 在编译阶段的汇编地址。

和inc 指令相对的是dec 指令，用于将目标操作数的内容减一，它们的指令格式相同，不再赘述。

源程序第37 行，正是loop 指令。就像我们刚才说的，它将CX 的内容减一，并判断是否为零。如果不为零，则跳转到标号digit 所在的位置处执行。

很显然，在指令的地址部分使用寄存器，而不是数值或者标号（其实标号是数值的等价形式，在编译后也是数值）有一个明显的好处，那就是可以在循环体里方便地改变偏移地址，如果使用数值就不能做到这一点。

检测点6.2

选择题：下面哪些指令是错误的，为什么？

A.add ax,[bx]　　B.mov ax,[si]　　C.mov ax,[cx]　　D.mov dx,[di]

E.mov dx,[ax]   F.inc byte [di]   G.div word [bx]

6.7　计算机中的负数

6.7.1　无符号数和有符号数

为了讲解后面的内容时能够顺利一些，现在我们离开源程序，来介绍一些题外的知识。

从本书的开篇到现在，我们一直没有提到负数，就好像世界上根本没有负数一样。计算机当然要处理负数，要不然它将没有多少实用价值。

在计算机中使用负数，这是一个容易令人产生迷惑的话题。不信？现在就开始了。

尽管我们从来没有考虑过数的正负问题，但是，事实上，我们在编写程序的时候，既可以使用正数，也可以使用负数。如图6-3 所示，我们在程序中用伪指令db 声明了一些正数和一些负数。

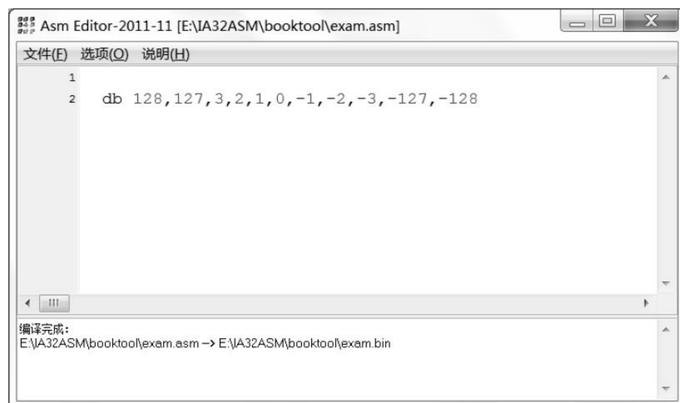


图6-3　在汇编源程序中使用负数的例子

图6-4 显示了编译后的结果。用伪指令db 声明的数据都只有一个字节的长度， 所以很容易在这两幅图的各个数之间建立对应关系。

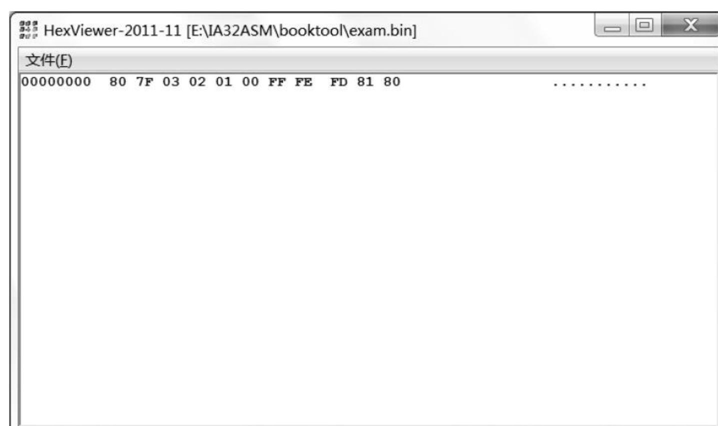


图6-4 正数和负数编译后的结果

前面的正数都很好理解，十进制数128 对应的二进制数是10000000，对应的十六进制数是0x80；十进制数0 对应的二进制数是00000000，对应的十六进制数是0x00。为什么我们对此不感到新鲜？因为这显得非常自然，从本书一开始到现在，我们就是这样工作的。

真正的麻烦在于后面的负数，比如－1，它在编译的时候，编译器会怎么做呢？

它很笨，但也很聪明。因为－1 其实等于0－1，它就知道可以做一次减法。当然，这个减法，不是你已经熟悉的十进制减法，这没有用，你得做二进制的减法，也就是用二进制数0 减去二进制数1，结果是

```
…1111111111111111111111111111111111
```

注意左边的省略号，这是因为在相减的过程中，不停地向左边借位的结果。因此，可以说，这个数字是很长的，取决于你什么时候停止借位。

再比如十进制数－2，可以用0－2 来得到，在二进制的世界里，该减法是二进制数0 减去二进制数10，结果是

```
…1111111111111111111111111111111110
```

同样，相减的过程要向左借位，所以这个数字相当长。但是，最右边那一位是0。

在计算机中，数字保存在寄存器里，而在16 位处理器里，寄存器通常是8 位和16 位的。因此，以上相减的结果，只能保留最右边的8 位或

者16 位。举个例子，十进制数－1 在寄存器AL中的二进制形式是

```
11111111
```

即0xFF；十进制数－2 在寄存器AL 中的二进制形式是

```
11111110
```

即0xFE。如果是16 位的寄存器，则相应地，要保留相减结果的最右边16 位。因此，十进制数-1 在AX 寄存器中的二进制形式是

```
1111111111111111
```

即0xFFFF；十进制数－2 在寄存器AX 中的二进制形式是

```
1111111111111110
```

即0xFFFE。

当然，数据还可以保存在内存中，或者编译后的二进制文件中。在二进制文件中，数据是用伪指令db 或者dw 等定义的。但是，数据的表示形式和它们在寄存器中的形式相同，以下代码片断很清楚地说明了这一点。

```
data0  db  -1       ;初始化为 0xFF
data1  db  -2       ;初始化为 0xFE
data2  dw  -1       ;初始化为 0xFFFF
data3  dw  -2       ;初始化为 0xFFFE
```

这是很令人吃惊的。因为我们知道，0xFF 等于十进制数255，但现在它又是十进制数－1，哪一个才是正确的呢？我们应该以哪一个为准呢？

好吧，假设这勉强能接受的话，那么，对照一下图6-3 和图6-4，你会发现，0x80 既是十进制数128，又是十进制数－128，到底哪一个是正确的呢？

这真是令人头疼的问题，不单单是对我们，对几十年前那些计算机工程师们来说也是如此。

一个良好的解决方案是，将计算机中的数分成两大类：无符号数和有符号数。无符号数的意思是我们不关心这些数的符号，因此也就无所谓正负，反正它们就是数而已，就像小学生一样，眼中只有自然数。在8位的字节运算中，无符号数的范围是00000000～11111111，即十进制的0～255；在16 位的字运算中，无符号数的范围是0000000000000000～1111111111111111，即十进制的0～65535；在将来要讲到的32 位运算中，无符号数的范围是00000000000000000000000000000000～11111111111111111111111111111111，即十进制的0～4294967295。很显然，我们以前使用的一直是无符号数。

相反地，有符号数是分正、负的，而且规定，数的正负要通过它的最高位来辨别。如果最高位是0，它就是正数；如果是1，就是负数。如此一来，在8 位的字节运算环境中，正数的范围是00000000～01111111，即十进制的0～127；负数的范围是10000000～11111111，即十进制的－128～－1。

正的有符号数，和与它同值的无符号数相同，这没什么好说的，毕竟它们形式上相同，按相同的方式处理最为方便。但是，负数就不同了，在这里，10000000～11111111 这些负数，都是用0减去它们相对应的正数得到的。想知道它们各自对应的正数是谁吗？很简单，因为“负数的负数”是正数，所以只需要用0 减去这个负数就行。所以，你可以试试看，因为

```
00000000－10000000＝10000000（十进制数 128）
00000000－11111111＝00000001（十进制数 1）
```

所以，10000000～11111111 这个范围内的有符号数，对应着十进制数－128～－1。

顺便说一下，在8086 处理器中，有一条指令专门做这件事，它就是neg。neg 指令带有一个操作数，可以是8 位或者16 位的寄存器，或者内存单元。如

```
neg al
neg dx
```

```
neg word [label_a]
```

它的功能很简单，用0 减去指令中指定的操作数。例子：如果AL 中的内容是00001000（十进制数8），执行neg al 后，AL 中的内容变为11111000（十进制数－8）；如果AL 中的内容为11000100（十进制数－60），执行neg al 后，AL 中的内容为00111100（十进制数60）。

相应地，在16 位的字运算环境中，正数的范围是0000000000000000～0111111111111111，即十进制的0～32767，负数的范围是1000000000000000～1111111111111111，即十进制的－32768～－1。

不要给计算机和编译器添麻烦。既然你已经知道一个字节可以容纳的数据范围是十进制的－128～127，就不要这样写：

```
mov al,-200
```

寄存器AL 只有8 位，因此，编译后，－200 将被截断，机器码为B0 38。你可以这样写：

```
mov ax,-200
```

这时，编译后的机器码为B8 38 FF。

同样的规则也适用于伪指令db 和dw。举例（以下均为十进制数）：

```
db 255                ;正确，可以看成声明无符号数
db -125               ;正确，数据未超范围
db -240               ;错误，超过字节所能容纳的数据范围，会被截断
dw -240               ;正确，数据未超范围
dw -30001             ;正确，数据未超范围
```

32 位有符号数是16 位和8 位有符号数的超集，16 位有符号数又是8 位有符号数的超集，它们互相之间有重叠的部分。正数还好说，十进制数15，在8 位运算环境中是00001111，在16 位运算环境中是0000000000001111，没有什么区别。但是，同一个负数，其表现形式略有差别。比如十进制数－3，它在8 位运算中是11111101，即0xFD；在16 位运算中，则是1111111111111101，即0xFFFD。这种差别的来源很简单，我们已经讲过了，在计算机中，－3 是用0 减去3 得到的，在8 位运算中只能保留结果的低8 位，即11111101（0xFD）；在16 位运算中只能保留结果的低16 位，即1111111111111101（0xFFFD）。

很显然，一个8 位的有符号数，要想用16 位的形式来表示，只需将其最高位，也就是用来辨别符号的那一位（几乎所有的书上都称之为符号位，实际上这并不严谨），扩展到高8 位即可。为了方便，处理器专门设计了两条指令来做这件事：cbw（Convert Byte to Word）和cwd（Convert Word to Double-word）。

cbw 没有操作数，操作码为98。它的功能是，将寄存器AL 中的有符号数扩展到整个AX。举个例子，如果AL 中的内容为01001111，那么执行该指令后，AX 中的内容为0000000001001111；如果AL 中的内容为10001101，执行该指令后，AX 中的内容为1111111110001101。

cwd 也没有操作数，操作码为99。它的功能是，将寄存器AX 中的有符号数扩展到DX:AX。举个例子，如果AX 中的内容为0100111101111001，那么执行该指令后，DX 中的内容为0000000000000000，AX 中的内容不变；如果AX 中的内容为1000110110001011，那么执行该指令后，DX 中的内容为1111111111111111，AX 中的内容同样不变。

尽管有符号数的最高位通常称为符号位，但并不意味着它仅仅用来表示正负号。事实上，通过上面的讲述和实例可以看出，它既是数的一部分，和其他比特一起共同表示数的大小，同时又用来判断数的正负。

6.7.2　处理器视角中的数据类型

无符号数和有符号数的划分并没有从根本上打消我们的疑虑，即假如寄存器AX 中的内容是0xB23C，那么，它到底是无符号数45628 呢，还是应当将其看成是－19908？

答案是，这是你自己的事，取决于你怎么看待它。对于处理器的多数指令来说，执行的结果和操作数的类型没有关系。换句话说，无论你是从无符号数的角度来看，还是从有符号数的角度来看，指令的执行结果都是正确无误的。比如

```
mov ah,al
```

这条指令显然根本不考虑操作数的类型。再比如

```
mov ah,0xf0
inc ah
```

在这里，0xf0 的二进制形式是11110000，它既可以解释为无符号数240（十进制），也可以解释为有符号数－16，毕竟它的符号位是1。无论如何，inc 是加一指令，这条指令执行后，AH 中的内容是二进制数11110001，既是无符号数241，也是有符号数－15。

再考虑加法运算。比如

```
mov ax,0x8c03
add ax,0x05
```

0x8c03 的二进制形式是1000110000000011，既可以看做无符号数35843（十进制），也可以看成是有符号数－29693（十进制）。在运算过程中，数的视角要统一，如果把0x8c03 看成是无符号数，那么0x05也是无符号数；如果0x8c03 是有符号数，那么0x05 也是有符号数。

关键是运算后的结果。很幸运的是，add 指令同样适用于无符号数和有符号数。所以，这两条指令执行后，AX 中的内容是0x8c08，分别可以看成是无符号数35848 和有符号数－29688。

再来考虑一下减法。考虑一下，如果要计算10－3，这其实可以看成是10＋（－3）。因此，使用以下三条指令就可以完成减法运算：

```
mov ah,10
mov al,-3
add ah,al
```

正是因为这个原因，很多处理器内部不构造减法电路，而是使用加法电路来做减法。

尽管如此，为了方便起见，处理器还是提供了减法指令sub，该指令和加法指令add 相似，目的操作数可以是8 位或者16 位通用寄存器，也可以是8 位或者16 位的内存单元；源操作数可以是通用寄存器，也可以是内存单元或者立即数（不允许两个操作数同时为内存单元）。比如

```
sub ah,al
sub dx,ax
sub [label_a],ch
```

因为处理器没有减法运算电路，所以，举例来说，sub ah,al 指令实际上等效于下面两条指令：

```
neg al
add ah,al
```

可以这么说，几乎所有的处理器指令既能操作无符号数，又能操作有符号数。但是，有几条指令除外，比如除法指令和乘法指令。

我们已经学过除法指令div。严格地说，它应该叫做无符号除法指令（Unsigned Divide），因为这条指令只能工作于无符号数。换句话说，只有从无符号数的角度来解释它的执行结果才能说得通。举个例子：

```
mov ax,0x0400
mov bl,0xf0
div bl                ;执行后，AL 中的内容为 0x04，即十进制数 4
```

从无符号数的角度来看，0x0400 等于十进制数1024，0xf0 等于十进制数240。相除后，寄存器AL 中的商为0x04，即十进制数4，完全正确。

但是，从有符号数的角度来看，0x0400 等于十进制数1024，0xf0 等于十进制数－16。理论上，相除后，寄存器AL 中结果应当是0xc0。因其最高位是“1”，故为负数，即十进制数为－64。

为了解决这个问题，处理器专门提供了一个有符号数除法指令idiv（Signed Divide）。idiv 的指令格式和div 相同，除了它是专门用于计算有符号数的。如果你决定要进行有符号数的计算，必须采用如下代码：

```
mov ax,0x0400
mov bl,0xf0
idiv bl                   ;执行后，AL 中的内容为 0xc0，即十进制数－64
```

在用idiv 指令做除法时，需要小心。比如用0xf0c0 除以0x10，也就是十进制数的除法－3904÷ 16。你的做法可能会是这样的：

```
mov ax,0xf0c0
mov bl,0x10
idiv bl
```

以上的代码是16 位二进制数除法，结果在寄存器AL 中。除法的结果应当是十进制数－244，遗憾的是，这样的结果超出了寄存器AL 所能

表示的范围，必然因为溢出而不正确。为此，你可能会用32 位的除法来代替以前的做法：

```
xor dx,dx              ;如此一来，DX：AX 中的数成了正数
mov ax,0xf0c0
mov bx,0x10
idiv bl
```

很遗憾，这依然是错的。十进制数－3904 的16 位二进制形式和32 位二进制形式是不同的。前者是0xf0c0，后者是0xffff0c0。还记得cwd 吗？你应该用这条指令把寄存器AX 中数的符号扩展到DX。所以，完全正确的写法是这样的：

```
mov ax,0xf0c0
cwd
mov bx,0x10
idiv bx
```

以上指令全部执行后，寄存器AX 中的内容为0xff0c，即十进制数－244。

主动权在你自己手上，在写程序的时候，你要做什么，什么目的，你自己最清楚。如果是无符号数计算，必须使用div 指令；如果你是在做有符号数计算，就应当使用idiv 指令。

检测点6.3

假如以下声明的是有符号数，那么，其中的负数是（　　　　　　　　　　　　　　）。

data0 db 0xf0,0x05,0x66,0xff,0x81

data1 dw 0xfff,0xffff,0x8b,0x8a08

6.8 数位的显示

一旦各个数位都分解出来了，下面的工作就是在屏幕上显示它们。源程序第40 行，将保存有各个数位的数据区首地址传送到基址寄存器BX。

一共有5 个数字要显示，它们在当前数据段内的起始偏移地址就是number 的汇编地址，且已传送到寄存器BX 中。为了依次得到这5 个数字，程序中使用的指令是

```
mov al,[bx+si]
```

在这里，我们的意图是，寄存器BX 中的内容是基地址，保持不变，当寄存器SI 的内容从0逐次增加到4，或者反过来，从4 递减到0 时，就可以通过BX+SI 来连续访问这5 个数字。在这里，SI 的作用相当于索引，因此它被称为索引寄存器（Index Register），或者叫变址寄存器。另一个常用的变址寄存器是DI。

注意，INTEL8086 处理器只允许以下几种基址寄存器和变址寄存器的组合：

```
[bx+si]
[bx+di]
[bp+si]
[bp+di]
```

这些组合可以用于任何带有内存操作数的指令中。其他任何组合，比如[bx+ax]、[cx+dx]、[ax+cx]等等，都是非法的。

因此，源程序第41 行，把初始的索引值4 传送到SI 寄存器，这是由于要先显示万位上的数字。

源程序第43 行，从指定的内存单元取出一个字节，传送到AL 寄存器，偏移地址是BX+SI。但是，它们之间的运算并非是在编译阶段进行的，而是在指令实际执行的时候，由处理器完成的。

源程序第44 行，将AL 中的数字加上0x30，以得到它对应的ASCII码。

源程序第45 行，将数字0x04 传送到寄存器AH。0x04 是显示属性，即前面讲过的黑底红字，无加亮，无闪烁。到此，AX 中是一个完整的字，前8 位是显示属性值，后8 位是字符的ASCII 码。

源程序第46 行，将AX 中的内容传送到由段寄存器ES 所指向的显示缓冲区中，偏移地址由DI 指定。还记得吗，在前面使用movsw 传送字符串“Label offset:”到显示缓冲区时，也使用了DI，当时DI 是指向显示缓冲区首地址的（0），而且每传送一次就自动加2。传送结束后，DI 正好指向字符“:”的下一个存储单元。之后，DI 一直没用过，还保持着原先的内容。

注意，如图6-5 所示，数据的传送是按低端字节序的，寄存器的低字节传送到显示缓冲区的低地址部分（字节），寄存器的高字节传送到显示缓冲区的高地址部分（字节）。

源程序第47 行，将DI 的内容加上2，以指向显示缓冲区的下一个单元。

源程序第48 行，将SI 的内容减1，使得下一次的BX+SI 指向千位数字。dec 是减一指令，和inc 指令一样，后面跟一个操作数，可以是8 位或者16 位的通用寄存器或者内存单元。

源程序第49 行，指令jns show 的意思是，如果未设置符号位，则转移到标号“show”所在的位置处执行。如图6-2 所示，Intel 处理器的标志寄存器里有符号位SF（Sign Flag），很多算术逻辑运算都会影响到该位，比如这里的dec 指令。如果计算结果的最高位是比特“0”，处理器把SF 位置“0”，否则SF 位置“1”。

处理器的任务是忠实地执行指令，多数时候，它不会知道你的意图，也不会知道你进行的是有符号数运算，还是无符号数运算。如果运算结果的最高位是“1”，它唯一所能做的，就是将SF标志置“1”，以示提醒，剩下的事，你自己看着办，它已经尽力了。

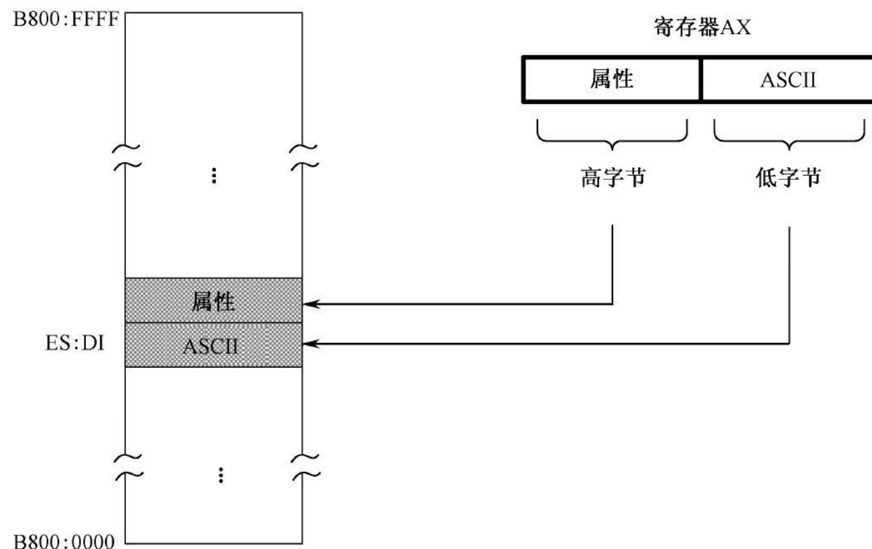


图6-5　低端字节序的字传送示意图

由于SI 的初始值为4，故第一次执行dec si 后，si 的内容为3，即二进制数0000000000000011，符号位是比特“0”，处理器将标志寄存器的SF 位清“0”。于是，当执行jns show 时，符合条件，于是转移到标号“show”所在的位置处执行，等于是开始显示下一个数位。

当显示完最后一个数位后，SI 的内容是零。执行dec si 指令后，由于产生了借位，实际的运算结果是0xffff（SI 只能容纳16 个比特），因其最高位是“1”，故处理器将标志位SF 置“1”，表明当前SI 中的结果可以理解为一个负数（－1）。于是，执行jns show 时，条件不满足，接着执行后面第51 行的指令。

jns 是条件转移指令，处理器在执行它的时候要参考标志寄存器的SF 位。除了只是在符合条件的时候才转移之外，它和jmp 指令很相似，它也是相对转移指令，编译后的机器指令操作数也是一个相对偏移量，是用标号处的汇编地址减去当前指令的汇编地址，再减去当前指令的长度得到的。

6.9 其他标志位和条件转移指令

在处理器内进行的很多算术逻辑运算，都会影响到标志寄存器的某些位。比如我们已经学过的加法指令add、逻辑运算指令xor 等。在下面的讲述中，请自行参考图6-2。

6.9.1 奇偶标志位PF

当运算结果出来后，如果最低8 位中，有偶数个为1 的比特，则PF=1；否则PF=0。例如：

```
mov ax,1000100100101110B        ;ax <- 0x892e
xor ax,3                        ;结果为 0x892d（1000100100101101B）
```

顺序执行以上两条指令后，因为结果是1000100100101101B，低8位是00101110B，有偶数个1，所以PF=1。

再如：

```
mov ah,00100110B                ;ah <- 0x26
mov al,10000001B                ;al <- 0x81
```

```
add ah,al                       ;ah <- 0xa7
```

以上，因为最后ah 的内容是0xa7（10100111B），包含奇数个1，故PF=0。

6.9.2 进位标志CF

当处理器进行算术操作时，如果最高位有向前进位或借位的情况发生，则CF=1；否则CF=0。比如：

```
mov al,10000000B             ;al <- 0x80
add al,al                     ;al <- 0x00
```

这里，寄存器AL 自己和自己做加法运算，并因为最高位是1 而产生进位。结果是，进位被丢弃，AL 中的最终结果为零。进位的产生，使得CF=1。同时，ZF=1，PF=1。

下面是因有借位而使得CF 为1 的例子：

```
mov ax,0
sub ax,1
```

CF 标志始终忠实地记录进位或者借位是否发生，但少数指令除外（如inc 和dec）。

6.9.3 溢出标志OF

对于无符号数运算来说，进位标志CF 通常意味着得到了错误的计算结果，因为目的操作数没能容纳那个进位。这里有一个例子：

```
mov ah,0xff
add ah,2          ;ah←0x01
```

执行以上两条指令后，进位标志CF 为1，这是肯定的了，因为最高位有进位。从无符号数的角度来看，是255+2，结果应当是257。但是你看，因为寄存器AH 只有8 位，所以进位丢失，得到的结果是1，这明显是错的。

但是，如果上面进行的是有符号数运算，那么，这实际上是在计算-1+2（十进制），AH 中的最终的结果是1，这是正确的。

很显然，同样的运算，从无符号数和有符号数的视角来看，是不同的。但是，在所有的情况下，处理器都不可能知道你的意图，不知道你进行的是有符号数运算，还是无符号数运算。为此，它提供了溢出标志OF，该标志的意思是，假定你进行的是有符号数运算，如果运算结果是正确的，那么OF=0，否则OF=1。比如上面的例子，因为从有符号数的角度来看，是-1 和2 相加，结果为1，未溢出，故OF=0。简单地说，OF 标志用于指示两个有符号数的运算结果是否错误。

再看一个例子：

```
mov ah,0x70
add ah,ah
```

首先，本次相加，用二进制数来说就是01110000+01110000=11100000，最高位没有进位，故CF=0。

其次，从无符号数的角度来看（十进制），即112+112=224，并未超出一个字节所能容纳的数值上限255，结果是正确的。

但是，从有符号数运算的角度来看（十进制），即112+112=-32，两正数相加，结果为负，明显是错的，在这种情况下，OF=1。错误的原因是，两个正数112 和112 相加，理论上的计算结果224 超出了寄存器AH所能容纳的有符号数的范围-128～127，所以破坏了符号位，使得结果变成了负数（-32）

既然如此，可以使用16 位寄存器AX，毕竟它能容纳的数据范围更大一些：

```
mov ax,0x70
add ax,ax
```

这次，无论它是有符号数运算，还是无符号数运算，结果都是正确的。故CF=0，OF=0。

因为在任何时候，处理器都不可能知道你的意图，不知道你进行的是有符号数运算，还是无符号数运算。因此，它所能做的，就是假定进行的是有符号数运算，并根据结果提供OF 标志，至于如何处理，是你自己的事。比如说，如果你进行的是无符号数运算，那么，你可以不用理会该标志。

6.9.4 现有指令对标志位的影响

由于是刚刚接触标志位，现将前面学过的指令对标志位的影响一一列举如下。在往后的学习中，但凡遇到新的指令时，除了讲解指令的功能和用法，也会说明其对标志位的影响。

注意，可以在Bochs 中察看标志位的状态，具体方法请参见本章后面的6.12.3 节。

add	OF、SF、ZF、AF、CF 和 PF 的状态依计算结果而定。
cbw	不影响任何标志位。
cld	DF=0，CF、OF、ZF、SF、AF 和 PF 未定义。未定义的意思是到目前为止还不打算让该指令影响到这些标志，因此，不要在程序中依赖这些标志。
cwd	不影响任何标志位。
dec	CF 标志不受影响，因为该指令通常在程序中用于循环计数，而且在循环体内通常有依赖 CF 标志的指令，故不希望它打扰 CF 标志；对 OF、SF、ZF、AF 和 PF 的影响依计算结果而定。
div/idiv	对 CF、OF、SF、ZF、AF 和 PF 的影响未定义。
inc	CF 标志不受影响，对 OF、SF、ZF、AF 和 PF 的影响依计算结果而定。
mov/movs	这类指令不影响任何标志位。
neg	如果操作数为 0，则 CF=0，否则 CF=1；对 OF、SF、ZF、AF 和 PF 的影响依计算结果而定。
std	DF=1，不影响其他标志位。
sub	对 OF、SF、ZF、AF、PF 和 CF 的影响依计算结果而定。
xor	OF=0，CF=0；对 SF、ZF 和 PF 依计算结果而定；对 AF 的影响未定义。

6.9.5 条件转移指令

“jcc”不是一条指令，而是一个指令族（簇），功能是根据某些条件进行转移，比如前面讲过的jns，意思是SF≠1（那就是SF=0 了）则转移。方便起见，处理器一般提供相反的指令，如js，意思是SF=1 则转移。爱上网的朋友们容易把它理解成“奸商”。

在汇编语言源代码里，条件转移指令的操作数是标号。编译成机器码后，操作数是一个立即数，是相对于目标指令的偏移量。在16 位处理器上，偏移量可以是8 位（短转移）或者16 位（相对近转移）。

相似地，jz 的意思是ZF 标志为1 则转移；jnz 的意思是ZF 标志不为1（为0）则转移。

jo 的意思是OF 标志为1 则转移，jno 的意思是OF 标志不为1（为0）则转移。

jc 的意思是CF 标志为1 则转移，jnc 的意思是CF 标志不为1（为0）则转移。

jp 的意思是PF 标志为1 则转移，jnp 的意思是PF 标志不为1（为0）则转移。爱上网的朋友们注意了，jp 可不是“极品”的意思。

转移指令必须出现在影响标志的指令之后，比如：

```
dec si
jns show
```

经验证明，像这种水到渠成的情况是很少的，多数时候，你会遇到一些和标志位关系不太明显的问题，比如，当AX 寄存器里的内容为0x30的时候转移，或者当AX 寄存器里的内容小于0xf0的时候转移，再或者，当AX 寄存器里的内容大于寄存器BX 里的内容时转移，这该怎么办呢？

好在处理器提供了比较指令cmp，它需要两个操作数，目的操作数可以是8 位或者16 位通用寄存器，也可以是8 位或者16 位内存单元；源操作数可以是与目的操作数宽度一致的通用寄存器、内存单元或者立即数，但两个操作数同时为内存单元的情况除外。比如：

```
cmp al,0x08
cmp dx,bx
cmp [label a],cx
```

cmp 指令在功能上和sub 指令相同，唯一不同之处在于，cmp 指令仅仅根据计算的结果设置相应的标志位，而不保留计算结果，因此也就不会改变两个操作数的原有内容。cmp 指令将会影响到CF、OF、SF、ZF、AF 和PF 标志位。

比较是拿目的操作数和源操作数比，重点关心的是目的操作数。拿指令cmp ax,bx 来说，我们关心的是AX 中的内容是否等于BX 中的内容，AX 中的内容是否大于BX 中的内容，AX 中的内容是否小于BX 中的内容，等等，AX 是被测量的对象，BX 是测量的基准。比较的结果如表6-1 所示。

表6-1　各种比较结果和相应的条件转移指令

比较结果	英文描述	指令	相关标志位的状态
等于	Equal	je	相减结果为零才成立，故要求 ZF＝1
不等于	Not Equal	jne	相减结果不为零才成立，故要求 ZF＝0
大于	Greater	jg	适用于有符号数比较 要求：ZF＝0（两个数不同，相减的结果不为零），并且 SF＝OF（如果相减后溢出，则结果必须是负数，说明目的操作数大；如果相减后未溢出，则结果必须是正数，也表明目的操作数大些）
大于等于	Greater or Equal	jge	适用于有符号数的比较 要求: SF＝OF
不大于	Not Greater	jng	适用于有符号数的比较 要求：ZF＝1（两个数相同，相减的结果为零），或者 SF≠OF（如果相减后溢出，则结果必须是正数，说明源操作数大；如果相减后未溢出，则结果必须是负数，同样表明源操作数大些）
不大于等于	Not Greater or Equal	jnge	适用于有符号数的比较 要求：SF≠OF
小于	Less	jl	适用于有符号数的比较，等同于“不大于等于” 要求：SF≠OF
小于等于	Less or Equal	jle	适用于有符号数的比较，等同于“不大于” 要求：ZF＝1（两个数相同，相减的结果为零），并且 SF≠OF（如果相减后溢出，则结果必须是正数，说明源操作数大；如果相减后未溢出，则结果必须是负数，同样表明源操作数大些）
不小于	Not Less	jnl	适用于有符号数的比较，等同于“大于等于” 要求：SF＝OF
不小于等于	Not Less or Equal	jnle	适用于有符号数的比较，等同于“大于” 要求：ZF＝0（两个数不同，相减的结果不为零），并且 SF＝OF（如果相减后溢出，则结果必须是负数，说明目的操作数大；如果相减后未溢出，则结果必须是正数，也表明目的操作数大些）

续表

比较结果	英文描述	指令	相关标志位的状态
高于	Above	ja	适用于无符号数的比较 要求：CF＝0（没有进位或借位）而且 ZF＝0（两个数不相同）
高于等于	Above or Equal	jae	适用于无符号数的比较 要求：CF＝0（目的操作数大些，不需要借位）
不高于	Not Above	jna	适用于无符号数的比较，等同于“低于等于”（见后） 要求：CF＝1 或者 ZF＝1
不高于等于	Not Above or Equal	jnae	适用于无符号数的比较，等同于“低于”（见后） 要求：CF＝1
低于	Below	jb	适用于无符号数的比较 要求：CF＝1
低于等于	Below or Equal	jbe	适用于无符号数的比较 要求：CF＝1 或者 ZF＝1
不低于	Not Below	jnb	适用于无符号数的比较，等同于“高于等于” 要求：CF＝0
不低于等于	Not Below or Equal	jnbe	适用于无符号数的比较，等同于“高于” 要求：CF＝0 而且 ZF＝0
校验为偶	Parity Even	jpe	要求：PF＝1
检验为奇	Parity Odd	jpo	要求：PF＝0

非常显而易见的是，如果你英语基础比较好，认识上面那些单词的话，这些指令都可以在短时间内轻松记住。英语基础不太好的人也不要灰心，事实上，根本不需要记住这些指令和它们的测试条件，因为我们平时很少用得了这么多。需要的时候再回过头来查查，这是个好办法，时间一长，自然就记住了。

最后一个要讲述的条件转移指令是jcxz（jump if CX is zero），意思是当CX 寄存器的内容为零时则转移。执行这条指令时，处理器先测试寄存器CX 是否为零。例如：

```
jcxz show
```

这里，“show”是程序中的一个标号。执行这条指令时，如果CX 寄存器的内容为零，则转移；否则不转移，继续往下执行。

检测点6.4

1．ZF 标志位和与该标志位有关的条件转移指令用得非常频繁，但很多人容易在ZF 标志位上犯糊涂，以为计算结果为零时，ZF 为“0”。为了证明你不糊涂，请填空：当ZF=（ ）时，表明计算结果为零；jz 指令的意思是当ZF=（ ），即计算结果为（ ）时转移；je 指令的意思是当ZF=（ ），即计算结果为（ ）时转移；jnz 指令的意思是当ZF=（ ），即计算结果不为（ ）时转移；jne 指令的意思是当ZF=（ ），即计算结果不为（ ）时转移。

2．写一小段程序，先比较寄存器AX 和BX 中的数值，然后，当AX 的内容大于BX 的内容时，转移到标号lbb 处执行；AX 的内容等于AX 的内容时，转移到标lbz 处执行；AX 的内容小于BX 的内容时，转移到标号lbl 处执行。

6.10 NASM 编译器的$和$$标记

源程序第51 行，用于在显示了各个数位之后，再显示一个字符“D”。目的地址是由ES:DI 给出的，源操作数是立即数0x0744，其中，高字节0x07 是黑底白字的显示属性，低字节0x44 是字符“D”的ASCII 码。字的写入是按低端字节序的，请自行参照图6-5。

整个程序到此结束。为了使处理器还有事做，源程序第53 行，是一个无限循环。NASM 编译器提供了一个标记“$”，该标记等同于标号，你可以把它看成是一个隐藏在当前行行首的标号。因此，jmp near $的意思是，转移到当前指令继续执行，它和

```
infi: jmp near infi
```

是一样的，没有区别，但不需要使用标号，更不必为给标号起一个有意义的名字而伤脑筋。

和第5 章一样，为了得到不多不少，正好512 字节的编译结果，同时最后两个字节还必须是0x55 和0xAA，需要在所有指令的后面填充一些无用的数据。

源程序第55 行，用于重复伪指令“db 0”若干次。重复的次数是由510-($-$$)得到的，除去0x55 和0xAA 后，剩余的主引导扇区内容是510 字节；$是当前行的汇编地址；$$是NASM 编译器提供的另一个标记，代表当前汇编节（段）的起始汇编地址。当前程序没有定义节或段，就默认地自成一个汇编段，而且起始的汇编地址是0（程序起始处）。这样，用当前汇编地址减去程序开头的汇编地址（0），就是程序实体的大小。再用510 减去程序实体的大小，就是需要填充的字节数。

就像处理器把内存划分成逻辑上的分段一样，源程序也应当按段来组织，划分成独立的代码段、数据段等。从本书第8 章开始，将引入这方面的内容。

6.11 观察运行结果

编译本章的源程序，并用FixVhdWr 将编译后的二进制文件写入虚拟硬盘的主引导扇区，然后启动VirtualBox，观察运行后的结果。在你的程序无错的情况下，显示的效果应当如图6-6 所示。

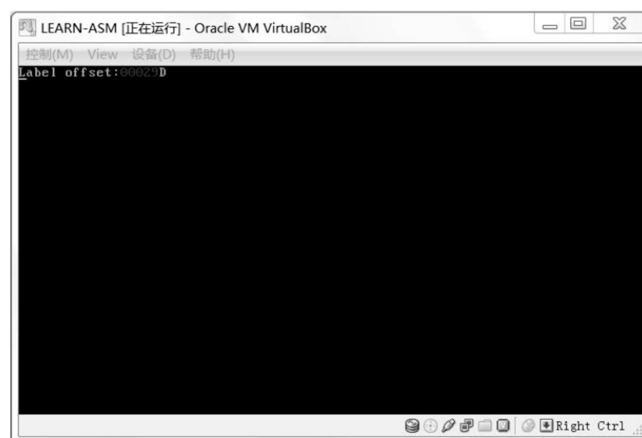


图6-6　本章程序的运行结果

6.12 本章程序的调试

6.12.1 调试命令“n”的使用

要调试本章的程序，可以利用上一章里介绍的方法，其中非常重要的一个调试命令是单步执行命令“s”。

单步执行有一个缺点，就是会陷入同一条指令的多次重复执行里，比如rep movsw 指令。如图6-7 所示，由于是在两个内存区域之间复制字符，rep movsw 指令要执行很多次，每当输入“s”命令后，执行的依然是movsw 指令，直到寄存器CX 的内容为零，复制过程结束后，才开始单步执行下一条指令。注意，在图中，Bochs 使用了rep movsw 指令的另一种形式“rep movsw word ptr es:[di],word ptr ds:[si]”，它们其实是一回事。

```
Bochs for Windows - Console
<bochs:10> s
Next at t=17825041
(0) [0x0000000000007c33] 0000:7c33 (unk. ctxt): mov cx, 0x000d            ; b90d
00
<bochs:11> s
Next at t=17825042
(0) [0x0000000000007c36] 0000:7c36 (unk. ctxt): rep movsw word ptr es:[di], word
 ptr ds:[si] ; f3a5
<bochs:12> s
Next at t=17825043
(0) [0x0000000000007c36] 0000:7c36 (unk. ctxt): rep movsw word ptr es:[di], word
 ptr ds:[si] ; f3a5
<bochs:13> s
Next at t=17825044
(0) [0x0000000000007c36] 0000:7c36 (unk. ctxt): rep movsw word ptr es:[di], word
 ptr ds:[si] ; f3a5
<bochs:14> s
Next at t=17825045
(0) [0x0000000000007c36] 0000:7c36 (unk. ctxt): rep movsw word ptr es:[di], word
 ptr ds:[si] ; f3a5
<bochs:15> s
Next at t=17825046
(0) [0x0000000000007c36] 0000:7c36 (unk. ctxt): rep movsw word ptr es:[di], word
 ptr ds:[si] ; f3a5
<bochs:16>
```

图6-7　单步执行rep movsw 指令时的情景

除了rep movsw 指令，本章中的loop 指令也会使单步执行陷入循环体中，直到循环条件不成立，退出循环时，才开始单步执行循环体外的下一条指令。如图6-8 所示，当单步执行循环指令loop .-9 时（本指令的物理内存地址是0x0000000000007C4A），下一条指令马上变成循环体内的第一条指令（xor dx,dx，物理内存地址为0x0000000000007C43）。只有当寄存器CX 的内容为零时，才开始单步执行循环体外的下一条指令。

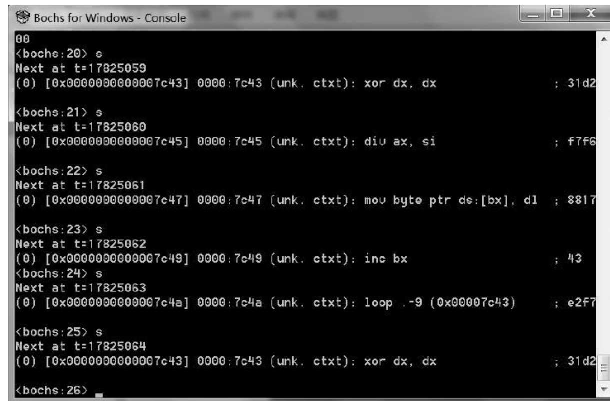


图6-8　Bochs 单步执行loop 指令时的情景

在图中，loop 指令的目标地址是用标号“.-9”表示的，这条指令就是本章程序中的指令

```
        loop digit
```

但是，程序在编译后，所有标号都消失了。当Bochs 重现这些程序时，不可能知道这里原先是一个标号“digit”。因此，它用loop 指令的操作数作为标号。我们知道，loop 指令的操作数是一个相对量，是用目标处的汇编（偏移）地址减去当前指令的汇编（偏移）地址，再减去loop 指令的长度（2）得到的。因此，如图中所示，0x7c4a 减去0x7c43（循环体内的第一条指令xor dx,dx），再减去2，只保留一个字节，就是0xf7，即十进制数－9。

可以想象，如果循环的次数很多（有时候，循环成千上万次是很正常的），则我们就无法调试循环体后面的程序。在这种情况下，你应当在执行rep movsb、repmovsw 和loop 指令的时候，使用调试命令“n”。此时，Bochs 将自动完成循环过程，并在循环体外的下一条指令前停住。

6.12.2　调试命令“u”的使用

之所以能够使用调试命令“n”来越过循环体，是因为Bochs 知道控制循环次数的是寄存器CX，它可以自动监视整个循环过程。

但是，“n”命令对于下面的循环结构无效：

```
show:
        mov al,[bx+si]
        add al,0x30
        mov ah,0x04
        mov [es:di],ax
        add di,2
        dec si
        jns show
```

原因很简单，条件转移指令（在这里是jns）不是循环指令，转移到的目标位置一般位于前方（源程序的后面），而不是像这里一样，目标位置是先前已经执行过的指令，于是恰巧构成了一个特殊的循环。

因此，如图6-9 所示，当用“n”命令执行jns show（在图中显示的是jns .-15）后，下一条指令又变成物理地址为0x0000000000007c52 处的指令mov al,byte ptr ds:[bx+si]（即mov al,[bx+si]），因为SF 标志为“0”。

如何越过条件转移指令构造的特殊循环体，往后调试执行呢？要解决这个问题，只需要知道循环体后面那条指令的物理地址即可，这可以使用反汇编命令“u”。

反汇编的意思是根据机器指令代码生成可读的汇编语言指令，正好与汇编过程相反。“u”命令可以使用两个参数，第一个参数是跟在“/”后面的数字，指定反汇编出多少条指令；第二个参数用于指定一个内存地址，Bochs 从这里开始反汇编操作。

```
<bochs:32> n
Next at t=17825086
(0) [0x0000000000007c52] 0000:7c52 (unk. ctxt): mov al, byte ptr ds:[bx+si] ; 8a
00
<bochs:33> n
Next at t=17825087
(0) [0x0000000000007c54] 0000:7c54 (unk. ctxt): add al, 0x30            ; 0430

<bochs:34> n
Next at t=17825088
(0) [0x0000000000007c56] 0000:7c56 (unk. ctxt): mov ah, 0x04            ; b404

<bochs:35> n
Next at t=17825089
(0) [0x0000000000007c58] 0000:7c58 (unk. ctxt): mov word ptr es:[di], ax  ; 2689
05
<bochs:36> n
Next at t=17825090
(0) [0x0000000000007c5b] 0000:7c5b (unk. ctxt): add di, 0x0002          ; 83c7
02
<bochs:37> n
Next at t=17825091
(0) [0x0000000000007c5e] 0000:7c5e (unk. ctxt): dec si                  ; 4e
<bochs:38> n
Next at t=17825092
(0) [0x0000000000007c5f] 0000:7c5f (unk. ctxt): jns .-15 (0x00007c52)    ; 79f1

<bochs:39> n
Next at t=17825093
(0) [0x0000000000007c52] 0000:7c52 (unk. ctxt): mov al, byte ptr ds:[bx+si] ; 8a
00
<bochs:40>
```

图6-9　在Bochs 中用"n"命令执行jns 指令时的情景

如图6-10 所示，在jns .-15 指令执行前，用"u"命令反汇编。该命令指示从指令jns .-15 所在的地址处（0x0000000000007c5f）开始反汇编，而且只需得到2 条指令即可。注意，如果是从当前地址处开始反汇编，则地址参数可以省略。在这里，只需使用"u/2"即可。

命令下达后，Bochs 迅速做出回应，给出了两条指令，并显示了各自所在的物理地址。很显然，条件转移指令jns 之后的那条指令是mov word ptr es:[di],0x0744，也就是本章程序中的mov word [es:di],0x0744，其物理地址是0x7c61。

依然如图中所示，为了越过这个特殊的循环结构，首先使用"b"命令把0x7c61 设为断点，然后执行"c"命令来连续执行程序，直至发现已经处于断点位置。

```
<bochs:28> n
Next at t=17825089
(0) [0x0000000000007c58] 0000:7c58 (unk. ctxt): mov word ptr es:[di], ax  ; 2689
05
<bochs:29> n
Next at t=17825090
(0) [0x0000000000007c5b] 0000:7c5b (unk. ctxt): add di, 0x0002          ; 83c7
02
<bochs:30> n
Next at t=17825091
(0) [0x0000000000007c5e] 0000:7c5e (unk. ctxt): dec si                  ; 4e
<bochs:31> n
Next at t=17825092
(0) [0x0000000000007c5f] 0000:7c5f (unk. ctxt): jns .-15 (0x00007c52)    ; 79f1

<bochs:32> u/2 0x7c5f
00007c5f: (                    ): jns .-15                  ; 79f1
00007c61: (                    ): mov word ptr es:[di], 0x0744 ; 26c7054407
<bochs:33> b 0x7c61
<bochs:34> c
(0) Breakpoint 2, 0x0000000000007c61 in ?? ()
Next at t=17825121
(0) [0x0000000000007c61] 0000:7c61 (unk. ctxt): mov word ptr es:[di], 0x0744 ; 2
6c7054407
<bochs:35>
```

图6-10　使用反汇编命令显示指定地址处的指令

6.12.3　用调试命令“info”察看标志位

为了察看标志寄存器FLAGS 的状态（各个标志位），可以在Bochs 中使用命令“info”。使用该命令可以显示多种类型的处理器信息，显示标志寄存器的状态只是其功能之一。

为了显示标志寄存器的状态，可以使用“eflags”参数，即“info eflags”。INTEL8086 的标志寄存器是16 位的，称做FLAGS；在32 位处理器上，该标志寄存器做了扩展，达到了32 位，称做EFLAGS。因此，在Bochs 中，应当输入“info eflags”而不是“info flags”。

要察看标志寄存器的状态，应当在调试本章程序的过程中进行。如图6-11 所示，我们在执行第33 行的xor dx,dx 指令之前，察看一下标志寄存器的状态。

```
Bochs for Windows - Console
(0) [0x0000000000007c36] 0000:7c36 (unk. ctxt): rep movsw word ptr es:[di], word
 ptr ds:[si] ; f3a5
<bochs:12> n
Next at t=17825055
(0) [0x0000000000007c38] 0000:7c38 (unk. ctxt): mov ax, 0x001d          ; b81d
00
<bochs:13> n
Next at t=17825056
(0) [0x0000000000007c3b] 0000:7c3b (unk. ctxt): mov bx, ax              ; 89c3

<bochs:14> n
Next at t=17825057
(0) [0x0000000000007c3d] 0000:7c3d (unk. ctxt): mov cx, 0x0005          ; b905
00
<bochs:15> n
Next at t=17825058
(0) [0x0000000000007c40] 0000:7c40 (unk. ctxt): mov si, 0x000a          ; be0a
00
<bochs:16> n
Next at t=17825059
(0) [0x0000000000007c43] 0000:7c43 (unk. ctxt): xor dx, dx              ; 31d2

<bochs:17> info eflags
id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af pf cf
<bochs:18>
```

图6-11　在Bochs 中察看标志寄存器的状态

如图中所示，当命令输入之后，Bochs 显示一行古怪的文字作为回应，请允许我来解释一下这些东西都是什么。

首先，像“id、vip、vif、ac、vm、rf、nt、IOPL”这些标志，是32 位处理器才有的，现在不用管它们。

然后，“of”是溢出标志；“df”是方向标志；“if”和“tf”是和中断有关标志，第9 章才能讲到；“sf”是符号标志；“zf”是零标志；“af”是辅助进位标

志；“pf”是奇偶标志；“cf”是进位标志。

问题是，光显示标志的名称，怎么知道某个标志位是“0”还是“1”呢？很简单，如果显示的标志名称是小写的，那么，说明该标志为“0”；否则，该标志的状态为“1”。如图中所示，因为符号标志是大写的“SF”，因此，表明该标志当前的状态是“1”。

注意，我们现在关注的是当xor 指令执行后，标志寄存器的变化情况。接下来，我们单步执行xor dx,dx 指令，然后再显示一次标志寄存器的内容。

如图6-12 所示，该指令执行后，符号标志的名称变成小写，零标志和奇偶标志的名称变为大写。请你想一想，这是为什么？

```
Bochs for Windows - Console
<bochs:13> n
Next at t=17825056
(0) [0x0000000000007c3b] 0000:7c3b (unk. ctxt): mov bx, ax              ; 89c3

<bochs:14> n
Next at t=17825057
(0) [0x0000000000007c3d] 0000:7c3d (unk. ctxt): mov cx, 0x0005          ; b905
00
<bochs:15> n
Next at t=17825058
(0) [0x0000000000007c40] 0000:7c40 (unk. ctxt): mov si, 0x000a          ; be0a
00
<bochs:16> n
Next at t=17825059
(0) [0x0000000000007c43] 0000:7c43 (unk. ctxt): xor dx, dx              ; 31d2

<bochs:17> info eflags
id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af pf cf
<bochs:18> s
Next at t=17825060
(0) [0x0000000000007c45] 0000:7c45 (unk. ctxt): div ax, si              ; f7f6

<bochs:19> info eflags
id vip vif ac vm rf nt IOPL=0 of df if tf sf ZF af PF cf
<bochs:20>
```

图6-12　xor dx,dx 指令对标志寄存器的影响

检测点6.5

调试本章程序。要求：使用反汇编命令定位到源程序第53 行（jmp near $），然后在这里设置断点，并用“c”命令连续执行到该断点位置。注意，因为Bochs 会把非指令的数据也视为指令，这将有可能导致反汇编不正确。因此，要小心避开这些数据区，在Bochs 把物理地址0x7C00之后的数据（一大堆零）也反汇编成指令时，不要感到惊讶。

本 章 习 题

1．在某程序中声明和初始化了以下的有符号数。请问，正数和负数各有多少？

```
data1 db 0x05,0xff,0x80,0xf0,0x97,0x30
data2 dw 0x90,0xfff0,0xa0,0x1235,0x2f,0xc0,0xc5bc
```

2．如果可能的话，尝试编写一个主引导扇区程序来做上面的工作。

3．请问下面的循环将执行多少次：

```
       mov cx,0
delay: loop delay
```

第7章　比高斯更快的计算

7.1　从1 加到1.0　的故事

伟大的数学家高斯在9 岁那年，用很短的时间完成了从1 到100 的累加。那原本是老师给学生们出的难题，希望他们能老老实实地待在教室里。

高斯的方法很简单，他发现这是50 个101 的求和：100＋1、99＋2、98＋3、…、50＋51，于是他很快算出结果是101×50＝5050。从1 加到100，高斯发现了其中的规律，当然很快就能算出结果。但是计算机很蠢，它不懂什么规律，只能从1 老老实实地加到100。不过，它的强项就是速度，而且不怕麻烦，当高斯还在审题的时候，它就累加出结果了。

计算累加和对计算机来说是小菜一碟，而这也不是本章的目的。本章的目标是：

1．通过计算1 到100 的累加和，学习一种重要的数据结构——栈，了解处理器为访问栈提供了怎样的支持。

2．总结INTEL8086 处理器的寻址方式。

3．学习几个新的处理器指令，它们是or、and、push 和pop。

4．学习在Bochs 中调试程序时察看栈的方法。

7.2　代码清单7-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：7-1（主引导扇区程序）

源程序文件：c07_mbr.asm

7.3 显示字符串

源程序第8 行，声明并初始化了一串字符（字符串），它的最终用途是要显示在屏幕上。我们可以直接用单引号把一串字符围起来：

```
message db '1+2+3+...+100='
```

NASM 支持这样的做法，同前一章相比，以这种方法声明字符串显得更方便、更直接。在编译阶段，编译器将把它们拆开，以形成一个个单独的字节。

为了跳过没有指令的数据区，源程序第6 行是jmp near start 指令。

源程序第11～15 行用于初始化数据段寄存器DS 和附加段寄存器ES。

源程序第18～28 行同样用于显示字符串，但采用了不同的方法，首先是用索引寄存器SI 指向DS 段内待显示字符串的首地址，即标号“message”所代表的汇编地址。然后，再用另一个索引寄存器DI 指向ES 段内的偏移地址0 处，ES 是指向0xB800 段的。

字符串的显示需要依赖循环。本次采用的是循环指令loop。loop 指令的工作又依赖于CX 寄存器，所以，源程序第20 行，用于在编译阶段计算一个循环次数，该循环次数等于字符串的长度（字符个数）。

循环体是从源程序第22 行开始的。首先从数据段中，逻辑地址为DS:SI 的地方取得第一个字符，将其传送到逻辑地址ES:DI，后者指向显示缓冲区。

紧接着，源程序第24 行，将DI 的内容加一，以指向该字符在显示缓冲区内的属性字节；第25 行，在该位置写入属性值0x07，即黑底白字。

源程序第26、27 行，分别将寄存器SI 和DI 的内容加一，以指向源位置和目标位置的下一个单元。

源程序第28 行，执行循环。loop 指令在执行时先将CX 的内容减一，然后，处理器根据CX是否为零来决定是否开始下一轮循环。当CX 为0 的时候，说明所有的字符已经显示完毕。

7.4 计算1 到100的累加和

接下来就是计算1 到100 的累加和了。处理器还没有智能到可以理解题意的程度，具体的计算方法和计算步骤只能由人来给出。

要计算1 到100 的累加和，可以采取这样的办法：先将寄存器AX 清零，再用AX 的内容和1相加，结果在AX 中；接着，再用AX 的内容和2相加，结果依旧在AX 中，……，就这样一直加到100。

为此，源程序第31 行，用xor 指令将寄存器AX 清零；源程序第32行，将第一个被累加的数“1”传送到寄存器CX。

源程序第34 行就开始累加了，每次相加之后，源程序第35 行，将CX 的内容加一，以得到下一个将要累加的数。

源程序第36 行，将CX 的内容同100 进行比较，看是不是已经累加到100 了。如果小于等于100，则继续重复累加过程，如果大于100，就不再累加，直接往下执行。

最后，AX 中将得到最终的累加和。需要特别说明的是，AX 可以容纳的无符号数最大是65535，再大就不行了。由于我们已经知道最终的结果是5050，所以很放心地使用了寄存器AX。要是你从1 加到1000，就得考虑使用两个寄存器来计算了。

7.5　累加和各个数位的分解与显示

7.5.1　栈和栈段的初始化

得到了累加和之后，下面的工作是将它的各个数位分解出来，并准备在屏幕上显示，好让我们知道这个数到底是多少。

和前两章不同，分解出来的各个数位并不保存在数据段中，而保存在一个叫做栈的地方。

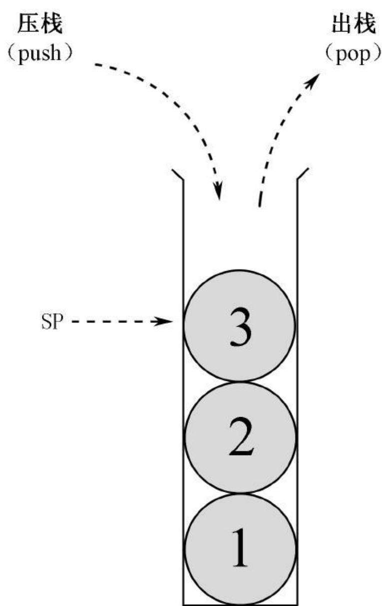


图7-1　一个说明栈工作原理的类比

栈（Stack）是一种特殊的数据存储结构，数据的存取只能从一端进行。这样，最先进去的数据只能最后出来，最后进去的数据倒是最先出来，这称为后进先出（Last In First Out，LIFO）。如图7-1 所示，可以把栈看成一个一端开口的塑料瓶，1 号球最先放进去，3 号球最后放进去，只能在3 号球和2 号球分别取出后，才能把1 号球取出来。

听起来像是在讲如何往盒子里放东西，或者从盒子里取东西。实际上，我们还是在讲内存，只不过是另一种特殊的读写方式而已。

和代码段、数据段和附加段一样，栈也被定义成一个内存段，叫栈段（Stack Segment），由段寄存器SS 指向。

针对栈的操作有两种，分别是将数据推进栈（push）和从栈中弹出数据（pop）。简单地说，就是压栈和出栈。压栈和出栈只能在一端进行，所以需要用栈指针寄存器SP （Stack Pointer）来指示下一个数据应当压入栈内的什么位置，或者数据从哪里出栈。

定义栈需要两个连续的步骤，即初始化段寄存器SS 和栈指针SP 的内容。源程序第40～42 行用于将栈段的段地址设置为0x0000，栈指针的内容设置为0x0000。

到目前为止，我们已经定义了3 个段，图7-2 是当前的内存布局。总的内存容量是1MB，物理地址的范围是0x00000～0xFFFFF，其中，假定数据段的长度是64KB（实际上它的长度无关紧要）， 占据了物理地址0x07C00 ～ 0x17BFF ， 对应的逻辑地址范围是0x07C0:0x0000 ～ 0x07C0:0xFFFF；代码段和栈段是同一个段，占据着物理地址0x00000 ～ 0x0FFFF ， 对 应 的 逻 辑 地 址 范 围 是 0x0000:0x0000 ～ 0x0000:0xFFFF。

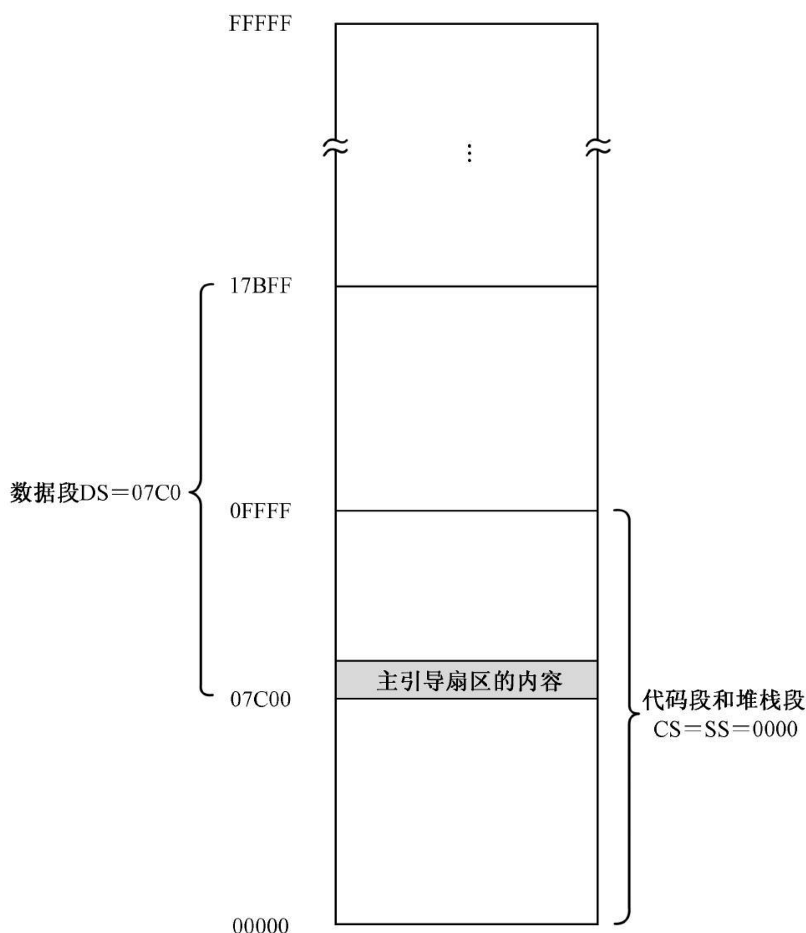


图7-2　本章程序的内存布局

虽然代码段和栈段在本质上指向同一块内存区域，但是不要担心，主引导程序只占据着中间的一小部分，我们有办法让它们互不干扰。

7.5.2　分解各个数位并压栈

数位的分解还是得靠做除法。源程序第44 行用于把除数10 传送到寄存器BX。

以往分解寄存器AX 中的数时，固定是分解5 次，得到5 个数位。但这也存在一个缺点，如果AX 中的数很小时，在屏幕上显示的数左边都是“0”，这当然是很别扭的。为此，本章的源程序做了改善，每次除法结束后，都做一次判断，如果商为0 的话，分解过程可以提前结束。

但是，由于每次得到的数位是压入栈的，将来还要反序从栈中弹出，为此，必须记住实际上到底有多少个数位。源程序第45 行，将寄存器CX 清零，并在后面的代码中用于累计有多少个数位。

源程序第47～53 行也是一个循环体，每执行一次，分解出一个数位。每次分解时，CX 加一，表明数位又多了一个，这是源程序第47 行所做的事。

源程序第48、49 行，将DX 清零，并和AX 一起形成32 位的被除数。

分解出的数位将来要显示在屏幕上，为了方便，源程序第50 行，直接将AL 中的商“加上”0x30，以得到该数字所对应的ASCII 码。

注意上一段话中的引号。这并不是真正的加法，or 并不是相加的指令，但由于此处的特殊情况，使得or 指令的执行结果和相加是一样的。

与xor 一样，or 也是逻辑运算指令。不同之处在于，or 执行的是逻辑“或”。数字逻辑中的“或”用于表示两个命题并列的情况。如果0 代表假，1 代表真，那么：

```
0 or 0 = 0
0 or 1 = 1
1 or 0 = 1
1 or 1 = 1
```

在处理器内部，or 指令的目的操作数可以是8 位或者16 位的通用寄存器，或者包含8/16位实际操作数的内存单元，源操作数可以是与目的操作数数据宽度相同的通用寄存器、内存单元或者立即数。比如：

```
or al,cl
or ax,dx
or [label_a],bx
or byte [bx],0x55
```

和其他指令一样，or 指令不允许目的操作数和源操作数都是内存单元的情况。当or 指令执行时，两个操作数相对应的比特之间分别进行各自的逻辑“或”运算，结果位于目的操作数中。举个例子，以下指令执行后，寄存器AL 中的内容是0xff。

```
mov al,0x55
or al,0xaa
```

再来看源程序第50 行，因为每次是除以10，所以在寄存器DL 中得到的余数，其高4 位必定为0。又由于0x30 的低4 位是0，高4 位是3，所以，DL 中的内容和0x30 执行逻辑“或”后，相当于是将DL 中的内容和0x30 相加。这是用逻辑“或”指令做加法的一个特例。

or 指令对标志寄存器的影响是：OF 和CF 位被清零，SF、ZF、PF 位的状态依计算结果而定，AF 位的状态未定义。

与or 相对应的是逻辑与“and”。如果0 代表假，1 代表真，那么

```
0 and 0 = 0
0 and 1 = 0
1 and 0 = 0
1 and 1 = 1
```

相应地，处理器设计了and 指令。在16 位处理器上，and 指令的两个操作数都应当是字节或者字。其中，目的操作数可以是通用寄存器和内存单元；源操作数可以是通用寄存器、内存单元或者立即数，但不允许两个操作数同时为内存单元，而且它们在数据宽度上应当一致。比如：

```
and al,0x55
and ch,cl
and ax,dx
and [label_a],ah
and word [bx],0xf0f0
and dx,[bx+si]
```

注意，“label_a”是一个标号，下同。

当这些指令执行时，两个操作数对应的各个比特位分别进行逻辑“与”，结果保存在目的操作数中。因此，下面的这些指令执行后，寄存器AX 中的结果是二进制数1000000000000100，即0x8004：

```
mov ax,1001_0111_0000_0100B
and ax,1000_0000_1111_0111B
```

and 指令执行后，OF 和CF 位被清零，SF、ZF、PF 位的状态依计算结果而定，AF 位的状态未定义。各个数位的ASCII 码是压入栈中的。源程序第51 行，push 指令的作用是将寄存器DX 的内容压入栈中。在16 位的处理器上，push 指令的操作数可以是16 位的寄存器或者内存单元。例如：

```
push ax
push word [label_a]
```

你可能觉得奇怪，push 指令只接受16 位的操作数，为什么要对内存操作数使用关键字“word”。事实上，8086 处理器只能压入一个字；但其后的32 位和64 位处理器允许压入字、双字或者四字，因此，关键字是必不可少的。

就8086 处理器来说，因为压入栈的内容必须是字，所以，下面的指令都是非法的：

```
push al
push byte [label_a]
```

处理器在执行push 指令时，首先将栈指针寄存器SP 的内容减去操作数的字长（以字节为单位的长度，在16 位处理器上是2），然后，把要压入栈的数据存放到逻辑地址SS:SP 所指向的内存位置（和其他段的读写一样，把栈段寄存器SS 的内容左移4 位，加上栈指针寄存器SP 提供的偏移地址）。

如图7-3 所示，代码段和栈段是同一个段，所以段寄存器CS 和SS 的内容都是0x0000。而且，栈指针寄存器SP 的内容在源程序第42 行被置为0。所以，当push 指令第一次执行时，SP 的内容减2，即0x0000－0x0002＝0xFFFE，借位被忽略。于是，被压入栈的数据，在内存中的位

置实际上是0x0000:0xFFFE。push 指令的操作数是字，而且Intel 处理器是使用低端字节序的，故低字节在低地址部分，高字节在高地址部分，正好占据了栈段的最高两个字节位置。

这只是第一次压栈操作时的情况。以后每次压栈时，SP 都要依次减2。很明显，不同于代码段，代码段在处理器上执行时，是由低地址端向高地址端推进的，而压栈操作则正好相反，是从高地址端向低地址端推进的。

push 指令不影响任何标志位。

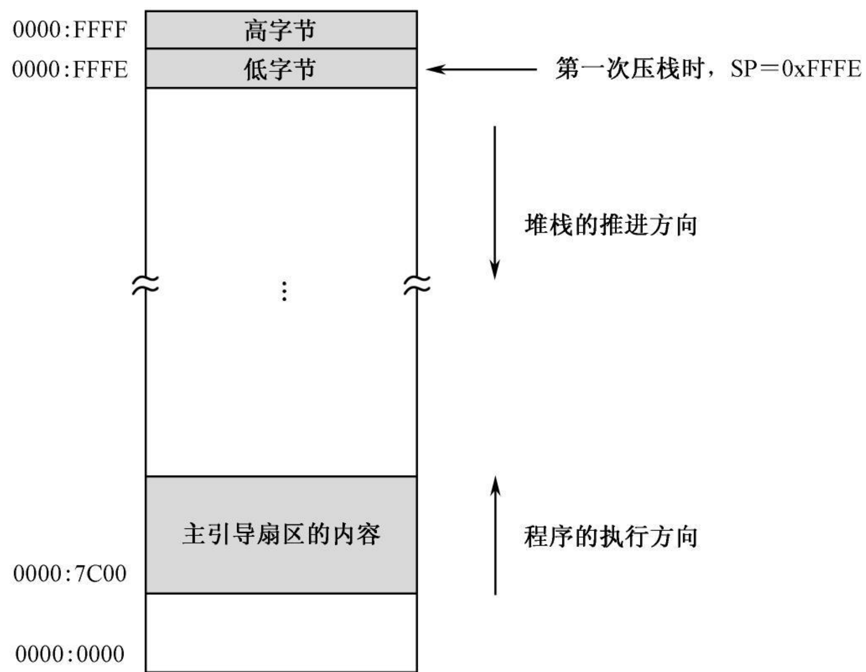


图7-3　第一次执行压栈操作时的内存状态

源程序第52、53 行，判断本次除法结束后，商是否为0。如果不为零，则再循环一次；如果为零，则表明不需要再继续分解了。

7.5.3　出栈并显示各个数位

压栈的次数（数位的个数）取决于AX 中的数有多大，位于寄存器CX 中。数位是按“个位”、“十位”、“百位”、“千位”、“万位”的顺序依次压栈的（实际情况取决于数的大小），出栈正好相反。所以，可以顺序将它们弹出栈并显示在屏幕上。

源程序第57 行，pop dx 指令的功能是将逻辑地址SS:SP 处的一个字弹出到寄存器DX 中，然后将SP 的内容加上操作数的字长（2）。

和push 指令一样，pop 指令的操作数可以是16 位的寄存器或者内存单元。例如：

```
pop ax
pop word [label_a]
```

pop 指令执行时，处理器将栈段寄存器SS 的内容左移4 位，再加上栈指针寄存器SP 的内容，形成20 位的物理地址访问内存，取得所需的数据。然后，将SP 的内容加操作数的字长，以指向下一个栈位置。

pop 指令不影响任何标志位。

源程序第58 行将弹出的数据写入显示缓冲区。索引寄存器DI 的内容是在前面显示字符串时用过的，期间一直没有改变过，它现在指向显示缓冲区中字符串之后的位置。

接着，源程序第59～61 行，将字符显示属性写入字符之后的单元，并再次递增DI 以指向显示缓冲区中下一个字符的位置。

源程序第62 行，每次执行loop 指令时，处理器都是先将寄存器CX 减一。当所有的数位都弹出和显示以后，CX 必定为零，这将导致退出循环。

当处理器最后一次执行出栈操作后，栈指针寄存器SP 的内容将恢复到最开始设置时的状态，即它的内容重新为0。

7.5.4　进一步认识栈

学习栈的知识，最好是先有一些感性认识，本章就是这么做的。现在，感性认识已经有了，剩下的，就是总结一下，做几点说明。

第一，push 指令的操作数可以是16 位寄存器或者16 位内存单元，push 指令执行后，压入栈中的仅仅是该寄存器或者内存单元里的数值，与该寄存器或内存单元不再相干。如果不理解这一点，就容易错误地以为压入了某个寄存器的值，比如AX 之后，将来还要再弹回AX 才行，这是不对的。所以，下面的指令是合法而且正确的：

```
push cs
pop ds
```

这两条指令的意思是，将代码段寄存器的内容压栈，并弹出到数据段寄存器DS。如此一来，代码段和数据段将属于同一个内存段。实际上，这两条指令的执行结果，和以下指令的执行结果相同：

```
mov ax,cx
mov ds,ax
```

第二，栈在本质上也只是普通的内存区域，之所以要用push 和pop 指令来访问，是因为你把它看成栈而已。实际上，如果你把它看成是普通的数据段而忘掉它是一个栈，那么它将不再神秘。

引入栈和push、pop 指令只是为了方便程序开发。临时保存一个数值到栈中，使用push 指令是最简洁、最省事的，但如果你不怕麻烦，可以不使用它。所以，下面的代码可以用来取代push ax 指令：

```
sub sp,2
mov bx,sp
mov [ss:bx],ax
```

同样，pop ax 指令的执行结果和下面的代码相同：

```
mov bx,sp
mov ax,[ss:bx]
add sp,2
```

你可能还有另一种想法，即，我连栈段都不用，SP 也省了，我自己把临时数据都保存在数据段中。好吧，如果是这样的话，你必须在数据段中开辟一些空间，并亲自维护一个指针来跟踪这些数据的存入和取出。当程序变得越来越复杂时，这些维护工作同样让你焦头烂额。

因此，显而易见的是，push 和pop 指令更方便，毕竟与栈访问有关的一切都是由处理器自动维护的。而且，总有一天你会发现，有些工作不使用栈来进行的话，是非常困难的。

第三，要注意保持栈平衡。如果在做某件事的时候要使用栈，那么，栈指针寄存器SP 在做这件事之前的值，应当和这件事做完后的值相

同。就是说，push 指令和pop 指令的数量应当是相同的。栈是反复使用的内存区域，如果使用不当，将会出现问题，下面就是一个例子：

```
repeat:
        push ax
        ......              ;其他非栈操作的指令
        pop bx
        pop ax
        loop repeat
```

以上的循环是干什么用的，做什么事情，这个不重要。因为每次循环时，都要用到寄存器AX 和BX 的原始内容，所以，循环体的开头要压栈保存它们，在循环体的末尾要出栈恢复它们。但是你看到了，由于一时疏忽，只压入了寄存器AX，而在出栈时，却多弹了一个数值到BX中。在这种情况下，栈是不平衡的，程序的运行结果当然也不会正确。

第四，在编写程序前，必须充分估计所需要的栈空间，以防止破坏有用的数据。特别是在栈段和其他段属于同一个段的时候。如图7-3 所示，栈段和代码段属于同一个内存段，段地址都是0x0000，段的长度都是64KB。主引导程序的长度是512（0x200）字节，从偏移地址0x7c00延伸到0x7e00。栈是向下增长的，它们之间有0xffff－0x7e00＋1＝0x8200 字节的空档。通常来说，我们的程序是安全的，因为不可能压入这么多的数据。但是，不能掉以轻心，栈定义得过小，而且程序编写不当，导致栈破坏了有用数据的情况也时有发生。

第五，尽管不能完全阻止程序中的错误，但是，通过将栈定义到一个单独的64KB 段，可以使错误仅局限于栈，而不破坏其他段的有用数据。假如栈的段地址是0x0000，大小是64KB，那么，无论SP 怎样变化，压栈和出栈操作始终会在该段内进行，而不会影响到其他无关的内存区域。这样，无论任何时候，即使是push 指令位于一个无限循环中，栈指针寄存器SP 的内容也永远只会在0x0000～0xFFFF 之间来回滚动，不会影响到其他内存段。

检测点7.1

1． 以下指令执行后，寄存器AX 中的内容是多少？

```
    mov ax,0xfff0
    and [data],ax
    or ax,[data]
data db 0x55,0xaa
```

2． 下面的说法中哪些是正确的？

A．8086 处理器执行压栈操作时，是先将SP 的内容减2，再访问栈段。

B．8086 处理器执行出栈操作时，是先将SP 的内容加2，再访问栈段。

C．如果SP 的内容为0xFFFC，则执行push ax 后，SP 的内容变为0xFFFA。

3． 在空白处补充指令或指令的操作数，使得程序可以把栈段当成数据段访问，并在寄存器DX 中得到AX 的压栈值。

```
push ds          ;保护本次操作之前的 DS
push bx          ;保护本次操作之前的 BX
push ax
mov bx,____
______,bx
mov bx,sp
___________
pop ax
pop bx           ;恢复本次操作之前的 BX
pop ds           ;恢复本次操作之前的 DS
```

7.6 程序的编译和运行

7.6.1 观察程序的运行结果

编译源程序7-1，然后将生成的二进制文件c07_mbr.bin 写入虚拟硬盘的主引导扇区，启动虚拟机观察程序运行结果。如果程序无误，结果应当如图7-4 所示。

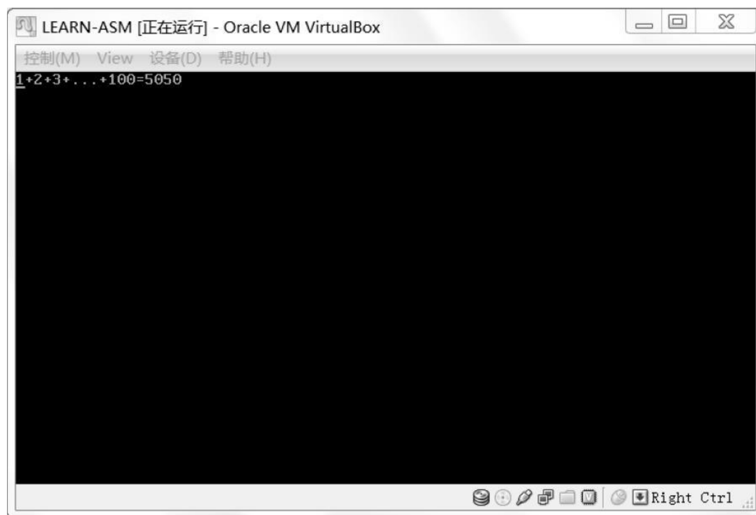


图7-4 本章程序在虚拟机中的运行结果

7.6.2 在调试过程中察看栈中内容

很多程序错误与栈的不当使用有关。因此，在调试程序的过程中，不可避免地要察看栈的状态，从中发现一些导致程序出错的蛛丝马迹。

在Bochs 中察看栈的命令是“print-stack”，它可以带一个参数，用于指定显示多少数据。如果不使用参数，则默认显示当前栈中的16 个字。

如图7-5 所示，当单步执行了“push dx”指令后，我们立即用“print-stack”命令来察看当前栈。当前栈是由段寄存器SS 指示的，栈顶是由栈指针寄存器SP 指示的。

```
Bochs for Windows - Console
Next at t=17825550
(0) [0x0000000000007c52] 0000:7c52 (unk. ctxt): push dx                ; 52
<bochs:3> s
Next at t=17825551
(0) [0x0000000000007c53] 0000:7c53 (unk. ctxt): cmp ax, 0x0000         ; 83f8
00
<bochs:4> print-stack
Stack address size 2
 | STACK 0xfffe [0x0030]
 | STACK 0x10000 [0x0000]
 | STACK 0x10002 [0x0000]
 | STACK 0x10004 [0x0000]
 | STACK 0x10006 [0x0000]
 | STACK 0x10008 [0x0000]
 | STACK 0x1000a [0x0000]
 | STACK 0x1000c [0x0000]
 | STACK 0x1000e [0x0000]
 | STACK 0x10010 [0x0000]
 | STACK 0x10012 [0x0000]
 | STACK 0x10014 [0x0000]
 | STACK 0x10016 [0x0000]
 | STACK 0x10018 [0x0000]
 | STACK 0x1001a [0x0000]
 | STACK 0x1001c [0x0000]
<bochs:5> _
```

图7-5　在Bochs 中察看当前栈中的数据

Bochs 并不知道栈的实际大小，因此，它只是显示栈顶（由SP 指示）以下的16 个字。如图中所示，栈顶数据是0x0030，其物理内存地址是0xFFFE，这是刚压入的寄存器DX 的内容。

因为栈是从高地址向低地址推进的，因此，所谓的“栈顶以下”，实际上指的是高地址方向。因此，下一个栈单元的物理内存地址是0xFFFE 加2，即0x10000。实际上，那里并不属于当前栈段，当前栈段的物理地址范围是0x00000~0x0FFFF，而且实际上我们只使用0x07E00~0x0FFFF之间的区域，但Bochs 并不知道这些。

7.7　8086处理器的寻址方式

处理器的一生，是忙碌的一生，只要它工作着，就必定是在取指令和执行指令。它就像勤劳的牛，吃的是电，挤出来的还是电，不过是另一种形式的电。多数指令操作的是数值。比如：

```
mov ax,0x55aa
```

这条指令执行时，把0x55aa 传送到寄存器AX。再如：

```
add dx,cx
```

这是把寄存器DX 中的数据和寄存器CX 中的数据相加，并把结果保留在DX 中，同时保持CX 中原有的内容不变。

所以，如果你问处理器整天忙什么，它一定会说："还能有什么，就是和数打交道！"

既然操作和处理的是数值，那么，必定涉及数值从哪里来，处理后送到哪里去，这称为寻址方式（Addressing Mode）。简单地说，寻址方式就是如何找到要操作的数据，以及如何找到存放操作结果的地方。

实际上，大多数的寻址方式我们都已经使用过，现在所做的只是一个完整的总结。当然，这里的讲解仅限于16 位的处理器。

7.7.1　寄存器寻址

最简单的寻址方式是寄存器寻址。就是说，指令执行时，操作的数位于寄存器中，可以从寄存器里取得。这种寻址方式的例子还是很多的，比如：

```
mov ax,cx
add bx,0xf000
inc dx
```

以上，第一条指令的两个操作数都是寄存器，是典型的寄存器寻址；第二条指令的目的操作数是寄存器，因此，该操作数也是寄存器寻

址；第三条指令就更不用说了。

7.7.2 立即寻址

立即寻址又叫立即数寻址。也就是说，指令的操作数是一个立即数。比如：

```
add bx,0xf000
mov dx,label_a
```

以上，第一条指令的目的操作数采用了寄存器寻址方式，用于提供被加数；第二个操作数（源操作数）用于给出加数0xf000。这是一个直接给出的数值，是立即在指令中给出的，最终参与加法运算的就是它，不需要通过其他方式寻找，故称为立即数。这也是一种寻址方式，称为立即寻址。

在第二条指令中，目的操作数也采用的是寄存器寻址方式。尽管源操作数是一个标号，但是，标号是数值的等价形式，代表了它所在位置的汇编地址。因此，在编译阶段，它会被转化为一个立即数。因此，该指令的源操作数也采用了立即寻址方式。

7.7.3 内存寻址

寄存器寻址的操作数位于寄存器中，立即寻址的操作数位于指令中，是指令的一部分。传统上，这是两种速度较快的寻址方式。但是，它们也有局限性。一方面，我们不可能总是知道要操作的数是多少，因此也就不可能总是在指令中使用立即数；另一方面，寄存器的数量有限，不可能总指望在寄存器之间来回倒腾。

考虑到内存容量巨大，所以，在指令中使用内存地址，来操作内存中的数据，是最理想不过了。正是因为内存访问如此重要，处理器才拥有好几种内存寻址方式。

我们知道，8086 处理器访问内存时，采用的是段地址左移4 位，然后加上偏移地址，来形成20位物理地址的模式，段地址由4 个段寄存器之一来提供，偏移地址要由指令来提供。

因此，所谓的内存寻址，实际上就是寻找偏移地址，这称为有效地址（Effective Address，EA）。换句话说，就是如何在指令中提供偏移地址，供处理器访问内存时使用。

1. 直接寻址

使用该寻址方式的操作数是一个偏移地址，而且给出了该偏移地址的具体数值。比如：

```
mov ax,[0x5c0f]
add word [0x0230],0x5000
xor byte [es:label_b],0x05
```

但凡是表示内存地址的，都必须用中括号括起来。

以上，在第一条指令中，源操作数使用的是直接寻址方式，当这条指令执行时，处理器将数据段寄存器DS 的内容左移4 位，加上这里的0x5c0f，形成20 位物理地址。接着，从该物理地址处取得一个字，传送到寄存器AX 中。

在第二条指令中，目的操作数采用的是直接寻址方式。当这条指令执行时，处理器用同样的方法，访问由段寄存器DS 指向的数据段，并把指令中的立即数加到该段中偏移地址为0x0230 的字单元里。

尽管在第三条指令中，目的操作数使用了标号和段超越前缀，但它依然属于直接寻址方式。原因很简单，标号是数值的等价形式，在指令编译阶段，会被转换成数值；而段超越前缀仅仅用来改变默认的数据段。

2. 基址寻址

很多时候，我们会有一大堆的数据要处理，而且它们通常都是挨在一起，顺序存放的。比如：

```
buffer dw 0x20,0x100,0x0f,0x300,0xff00
```

假如要将这些数据统统加一，那么，使用直接寻址的指令序列肯定是这样的：

```
inc word [buffer]
inc word [buffer+2]
inc word [buffer+4]
…
```

这样做好吗？当然，程序本身是没有问题的。但是，考虑到它的效率和代码的简洁性，特别是这些工作用循环来完成会更好，可以使用基址寻址。所谓基址寻址，就是在指令的地址部分使用基址寄存器BX 或者BP 来提供偏移地址。比如：

```
mov [bx],dx
```

```
add byte [bx],0x55
```

以上，第一条指令中的目的操作数采用了基址寻址。在指令执行时，处理器将数据段寄存器DS 的内容左移4 位，加上基址寄存器BX 中的内容，形成20 位的物理地址。然后，把寄存器DX中的内容传送到该地址处的字单元里。

第二条指令中的目的操作数也采用的是基址寻址。指令执行时，将数据段寄存器DS 的内容左移4 位，加上寄存器BX 中的内容，形成20 位的物理地址。然后，将指令中的立即数0x55 加到该地址处的字节单元里。

使用基址寻址可以使代码变得简洁高效。比如，可以用以下的代码来处理上面的批量加一任务：

```
    mov bx,buffer
    mov cx,4
lpinc:
    inc word [bx]
    add bx,2
    loop lpinc
```

基址寻址的寄存器也可以是BP。比如：

```
mov ax,[bp]
```

这条指令的源操作数采用了基址寻址方式。但是，与前面的指令相比，它稍微有些特殊。原因在于，它采用是基址寄存器BP，在形成20 位

的物理地址时，默认的段寄存器是SS。也就是说，它经常用于访问栈。这条指令执行时，处理器将栈段寄存器SS 的内容左移4 位，加上寄存器BP 的内容，形成20 位的物理地址，并将该地址处的一个字传送到寄存器AX 中。

我们知道，栈是后进先出的数据结构，访问栈的一般方法是使用push 和pop 指令。比如我们用以下的指令压入两个数据：

```
mov ax,0x5000
push ax
mov ax,0x7000
push ax
```

很显然，如果要用pop 指令弹出数据，就必须先弹出0x7000，才能弹出0x5000，除非你改变了栈指针SP 的内容，否则这个顺序是不可能改变的。

但是，有时候我们希望，而且必须得越过这种限制，去访问栈中的内容，还不能破坏栈的状态，特别是栈指针寄存器SP 的内容，使得push 和pop 操作能正常进行。一个典型的例子是高级语言里的函数调用，所有的参数都位于栈中。为了能访问到那些被压在栈底的参数，这时，BP 就能派上用场：

```
mov ax,0x5000
push ax
mov bp,sp
mov ax,0x7000
push ax
mov dx,[bp]          ;dx 中的内容为 0x5000
```

以上，在压入0x5000 之后，立即将栈指针SP 保存到BP。后面，尽管栈顶的数据0x7000 没有出栈，但依然可以用BP 取出压在栈下面的0x5000。如此一来，正常的push 和pop 操作照样进行，同时，还能访问到栈中的参数。

基址寻址允许在基址寄存器的基础上使用一个偏移量。有时候，这使得它更加灵活。比如：

```
mov dx,[bp-2]
```

处理器在执行时，将段寄存器SS 的内容左移4 位，加上BP 的内容，再减去偏移量2 以形成物理地址。这样一来，在保持基址寄存器BP 内容不变的情况下，就可以访问栈中的任何元素。这里，偏移量仅用于在指令执行时形成有效地址，不会改变寄存器BP 的原有内容。

这种增加偏移量的做法也适用于基址寄存器BX。以下代码是前面那个批量加一任务的新版本：

```
        xor bx,bx
        mov cx,4
lpinc:
        inc word [bx+buffer]
        add bx,2
        loop lpinc
```

以上代码和前一个版本相比，没有太大变化，区别仅仅在于，BX 现在是从0 开始递增的，inc 指令操作数的偏移地址由BX 和标号buffer 所代表的值相加得到。相加操作在指令执行时进行，仅用于形成有效偏移地址，不会影响到BX 寄存器的内容。

3．变址寻址

变址寻址类似于基址寻址，唯一不同之处在于这种寻址方式使用的是变址寄存器（或称索引寄存器）SI 和DI。例如：

```
mov [si],dx
add ax,[di]
xor word [si],0x8000
```

和基址寻址一样，当带有这种操作数的指令执行时，除非使用了段超越前缀，处理器会访问由段寄存器DS 指向的数据段，偏移地址由寄存器SI 或者DI 提供。

同样地，变址寻址方式也允许带一个偏移量：

```
mov [si+0x100],al
and byte [di+label_a],0x80
```

以上第二条指令中，尽管使用的是标号，但本质上属于一个编译阶段确定的数值。

4．基址变址寻址

让处理器支持多种寻址方式会增加硬件上的复杂性，但可以增强它的数据处理能力，这么做是值得的。说到数据处理，下面是一个稍微复杂一些的任务：

```
string db 'abcdefghijklmnopqrstuvwxyz'
```

以上声明了标号“string”并初始化了26 个字节的数据。现在，你的任务是，将这26 字节的数据在原地反向排列。

这个问题不难，所以你可能很快想到使用栈，先将这26 个数据压栈，再反向出栈，因为栈是后进先出的，正好符合要求。代码是这样的（代码段、栈段初始化的代码统统省略）：

```
        mov cx,26                ;循环次数，从 26 到 1，共 26 次
        mov bx,string            ;数据区首地址（基地址）
lppush:
```

```
        mov al,[bx]
        push ax
        inc bx
        loop lppush               ;循环压栈

        mov cx,26
        mov bx,string
lppop:
        pop ax
        mov [bx],al
        inc bx
        loop lppop                ;循环出栈
```

这的确是个好办法。不过，8086 处理器也支持一种基址加变址的寻址方式，简称基址变址寻址，可能用起来更方便。

使用基址变址的操作数可以使用一个基址寄存器（BX 或者BP），外加一个变址寄存器（SI 或者DI）。它的基本形式是这样的：

```
mov ax,[bx+si]
add word [bx+di],0x3000
```

以上，第一条指令的源操作数采用了基址变址寻址。当处理器执行这条指令时，把数据段寄存器DS 的内容左移4 位，加上基址寄存器BX的内容，再加上变址寄存器SI 的内容，共同形成20 位的物理地址。然后，从该地址处取得一个字，传送到寄存器AX 中。

第二条指令与第一条指令类似，只不过是加法指令，它的目的操作数采用了基址变址寻址，源操作数采用的是立即寻址。这条指令执行时，处理器访问由段寄存器DS 指向的数据段，加上由BX 和DI 相加形成的偏移地址，共同形成20 位的物理地址，然后将立即数0x3000 加到该地址处的字单元里。

采用基址变址寻址方式的排序代码如下：

```
        mov bx,string                ;数据区首地址
        mov si,0                     ;正向索引
        mov di,25                    ;反向索引
    order:
        mov ah,[bx+si]
        mov al,[bx+di]
        mov [bx+si],al
        mov [bx+di],ah               ;以上 4 行用于交换首尾数据
        inc si
        dec di
        cmp si,di
        jl order                     ;首尾没有相遇，或者没有超越，继续
```

和前面使用栈的代码相比，指令的数量没有明显减少，这说明任务还不够复杂，也许只能这么解释了。但是，它同样很方便，很有效，不是吗？

同样地，基址变址寻址允许在基址寄存器和变址寄存器的基础上带一个偏移量。比如：

```
    mov [bx+si+0x100],al
    and byte [bx+di+label_a],0x80
```

本 章 习 题

1．修改代码清单7-1 的第31～37 行，使用loop 指令来计算累加和。要求：CX 寄存器既用来控制循环次数，同时还用来作为被累加的数。

2．在16 位的处理器上，做加法的指令是add，但它每次只能做8 位或16 位的加法。除此之外，还有一个带进位加法指令adc（Add With Carry），它的指令格式和add 一样，目的操作数可以是8 位或16 位的通用寄存器和内存单元，源操作数可以是与目的操作数宽度一致的通用寄存器、内存单元和立即数（但目的操作数和源操作数同为内存单元的除外）。不过，adc 指令在执行的时候，除了将目的操作数和源操作数相加，还要加上当前标志寄存器的CF 位。也就是说，视CF 位的状态，还要再加0 或者加1。这样一来，用adc 指令配合add 指令，就可以计算16 位以上的加法。

adc 指令对OF、SF、ZF、AF、CF 和PF 的影响视计算结果而定。

现在，请编写一段主引导扇区程序，计算1 到1000 的累加和，并在屏幕上显示结果。

第8章　硬盘和显卡的访问与控制

总是把目光放在一个小小的主引导扇区上，这没什么意思。现在，是我们离开它，向自由天地迈进的时候了。但是，应该迈向哪里呢？

主引导扇区是处理器迈向广阔天地的第一块跳板。离开主引导扇区之后，前方通常就是操作系统的森林，也就是我们经常听说的DOS、Windows、Linux、UNIX 等。

操作系统也是由一大堆指令组成的，之所以将其比作“森林”，是因为它包含了更多的指令，也许是几万条、几十万条，甚至几千万条的指令。相比之下，我们在前面编写的那些指令代码则相形见绌了。

和主引导扇区程序一样，操作系统也位于硬盘上。操作系统是需要安装到硬盘上的，这个安装过程不但要把操作系统的指令和数据写入硬盘，通常还要更新主引导扇区的内容，好让这块跳板直接连着操作系统。不像我们，一直用主引导扇区来显示字符和做加法。

操作系统通常肩负着处理器管理、内存分配、程序加载、进程（即已经位于内存中的程序）调度、外围设备（显卡、硬盘、声卡等）的控制和管理等任务。举个例子来说，你每天都要使用的Windows，它可以让你看到计算机内都有几块硬盘，都安装了哪些程序（通过图标来显示），并允许你双击图标运行这些程序，这都是托了操作系统（Windows）的福。要不然的话，这都是不可能的事。

凭个人之力，写一个非常完善的操作系统，这几乎是不可能的事。但是，写个小程序，模拟一下它的某个功能，还是可以的。我们知道，编译好的程序通常都存放在像硬盘这样的载体上，需要加载到内存之后才能执行。这个过程并不简单，首先要读取硬盘，然后决定把它加载到内存的什么位置。最重要的是，程序通常是分段的，载入内存之后，还要重新计算段地址，这叫做段的重定位。

程序可以有千千万万个，但加载过程却是固定的。在本章，我们把主引导扇区改造成一个程序加载器，或者说是一个加载程序，它的功能是加载用户程序，并执行该程序（将处理器的控制权交给该程序）。总的说来，本章的目标是：

1． 模拟操作系统加载应用程序的过程，演示段的重定位方法，最终使你彻底理解8086 处理器的分段内存管理机制。

2． 学习x86 处理器过程调用的程序执行机制。

3． 以读硬盘扇区和控制屏幕光标为实例，了解x86 处理器访问外围硬件设备的方法。

4． 总结JMP 和CALL 指令的全部格式。

5． 认识更多的x86 处理器指令，如in、out、shl、shr、rol、ror、jmp、call、ret 等。

8.1 本章代码清单

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：8-1（主引导扇区程序/加载器），源程序文件：c08_mbr.asm

本章代码清单：8-2（被加载的用户程序），源程序文件：c08.asm

8.2 用户程序的结构

8.2.1 分段、段的汇编地址和段内汇编地址

处理器的工作模式是将内存分成逻辑上的段，指令的获取和数据的访问一律按“段地址：偏移地址”的方式进行。相对应地，一个规范的程序，应当包括代码段、数据段、附加段和栈段。这样一来，段的划分和段与段之间的界限在程序加载到内存之前就已经准备好了。

和我们以前编写的源程序不同，代码清单8-2 很长。当然，真正的不同之处在于，代码和数据是以段的形式组织的。当然，因为清单很长，看起来并不是非常明显。为了清楚起见，图8-1给出了整个源程序的组织结构。

NASM 编译器使用汇编指令“SECTION”或者“SEGMENT”来定义段。它的一般格式是

```
SECTION 段名称
```

或者

```
SEGMENT 段名称
```

每个段都要求给出名字，这就是段名称，它主要用来引用一个段，可以是任意名字，只要它们彼此之间不会重复和混淆。

NASM 编译器不关心段的用途，可能也根本不知道段的用途，不知道它是数据段，还是代码段，或是栈段。事实上，这都不重要，段只用来分隔程序中的不同内容。

不过，话又说回来了，作为程序员，每个段的用途，你自己是清楚的。所以，为每个段起一个直观好记的名字，那是应该的。如图8-1 所示，第一个段的名字是“header”，表明它是整个程序的开头部分；第二个段的名字是“code”，表明这是代码段；第三个段的名字是“data”，表明这是数据段。

比较重要的是，一旦定义段，那么，后面的内容就都属于该段，除非又出现了另一个段的定义。另外，如图8-2 所示，有时候，程序并不以段定义语句开始。在这种情况下，这些内容默认地自成一个段。最为典型的情况是，整个程序中都没有段定义语句。这时，整个程序自成一个段。

NASM 对段的数量没有限制。一些大的程序，可能拥有不止一个代码段和数据段。

Intel 处理器要求段在内存中的起始物理地址起码是16 字节对齐的。这句话的意思是，必须是16 的倍数，或者说该物理地址必须能被16 整除。

相应地，汇编语言源程序中定义的各个段，也有对齐方面的要求。具体做法是，在段定义中使用“align=”子句，用于指定某个SECTION 的汇编地址对齐方式。比如说，“align=16”就表示段是16 字节对齐的，“align=32”就表示段是32 字节对齐的。

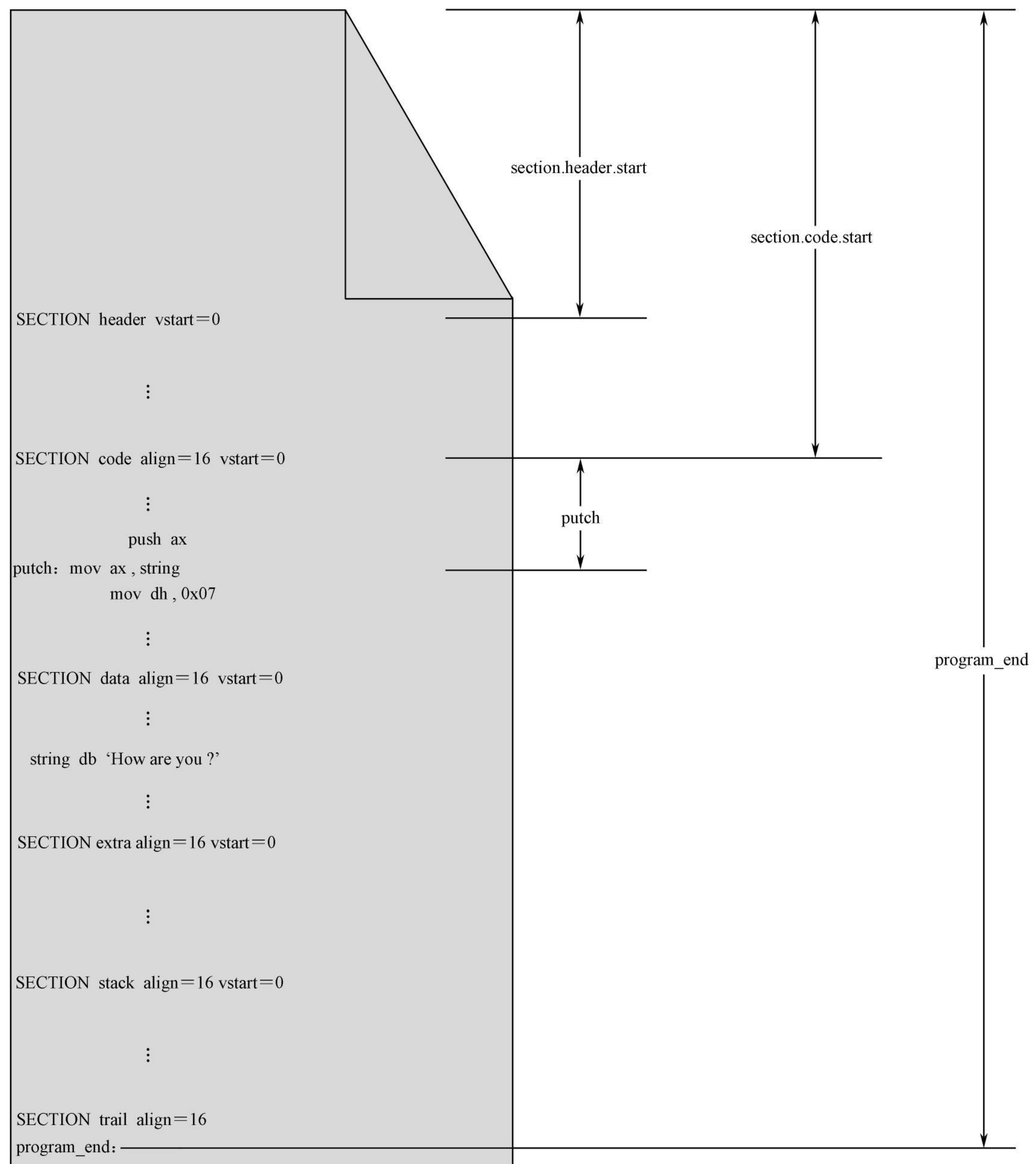


图8-1　用户程序的一般结构



图8-2　程序并非以段定义开始的情况

在源程序编译阶段，编译器将根据align 子句确定段的起始汇编地址。如图8-3 所示，这里定义了三个段，分别是data1、data2 和data3，每个段里只有一个字节的数据，分别是0x55、 0xaa 和0x99。

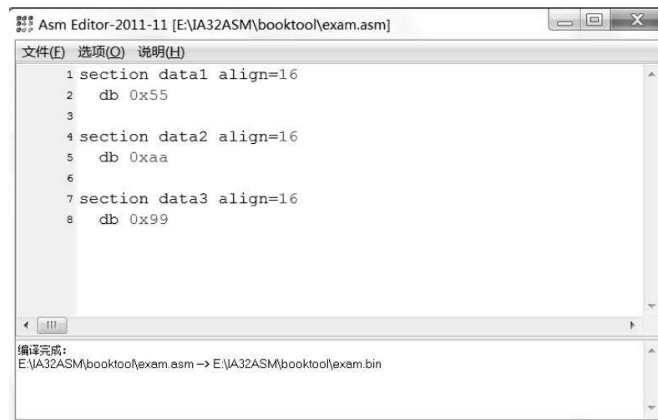


图8-3　align 子句对段的影响（编译之前的源代码）

理论上，如果不考虑段的对齐方式，那么段data1 的汇编地址是0，段data2 的汇编地址是1，段data3 的汇编地址是2。

但是，在这里，每个段的定义中都包含了要求16 字节对齐的子句，情况便不同了。如图8-4所示，这是编译后的结果，因为在段data1 之前没有任何内容，故段data1 的起始汇编地址是0（在图中是0x00000000），而且地址0 本身就是16 字节对齐的，符合align 子句的要求。

段的汇编地址其实就是段内第一个元素（数据、指令）的汇编地址。因此，在段data1 中声明和初始化的0x55 位于汇编地址0x00000000处。

段data2 也要求是16 字节对齐的。问题是，从汇编地址0x00000001 开始，只有0x00000010（十进制的16）才能被16 整除。于是，编译器将0x00000010 作为段data2 的汇编地址，并在两个段之间填充15 字节的0x00（段data1 只有1 字节的长度）。

段data3 的处理与前面两个段相同。因为段data2 只有1 字节，故也需要在它们之间填充15 字节。这样，段data3 的汇编地址就是0x00000020（十进制的32）。段data3 也只有1 字节（0x99），所以，汇编地址0x00000020 处是0x99，这也是编译结果中的最后一字节。

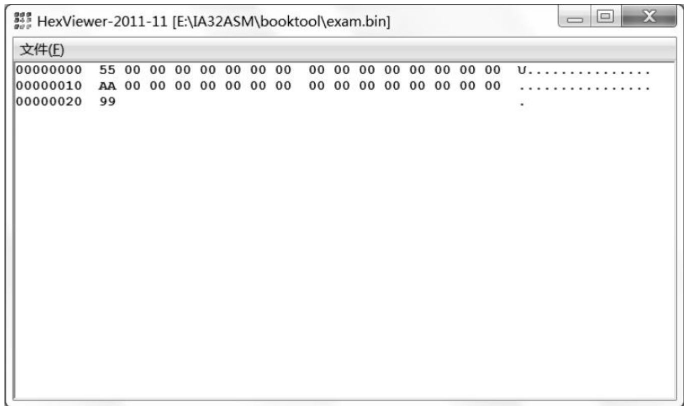


图8-4 align 子句对段的影响（编译之后的二进制文件）

正如我们刚刚讨论过的，每个段都有一个汇编地址，它是相对于整个程序开头（0）的。为了方便取得该段的汇编地址，NASM 编译器提供了以下的表达式，可以用在你的程序中：

```
section.段名称.start
```

如图8-1 所示，段“header”相对于整个程序开头的汇编地址是section.header.start，段“code”相对于整个程序开头的汇编地址是section.code.start。在这个例子中，因为段“header”是在程序的一开始定义的，它的前面没有其他内容，故section.header.start=0。

如图8-1 所示，段定义语句还可以包含“vstart=”子句。尽管定义了段，但是，引用某个标号时，该标号处的汇编地址依然是从整个程序的开头计算的，而不是从段的开头处计算的。

这就很麻烦（有时候也很有用）。因此，vstart 可以解决这个问题。如图8-1 所示，“putch”是段code 中的一个标号，原则上，该标号代表的汇编地址应该从程序开头计算。但是，因为段code的定义中有“vstart=0”

子句，所以，标号“putch”的汇编地址要从它所在段的开头计算，而且从0开始计算。

如图8-1 所示，同样的情形也出现在段data 中。段data 的定义中也有“vstart=0”子句，因此，当我们在段code 中引用段data 中的标号“string”时（mov ax,string），尽管在图中没有标明，标号“string”所代表的汇编地址是相对于其所在段data 的。也就是说，传送到寄存器AX 中的数值是标号string 相对于段data 起始处的长度。

但是，图中最后一个段trail 的定义中没有包含“vstart=0”子句。那就对不起了，该段内有一个标号“program_end”，它的汇编地址就要从整个程序开头计算。因为它是整个程序中的最后一行，从这个意义上来说，它所代表的汇编地址就是整个程序的大小（以字节计）。

检测点8.1

对于以下程序片断，假如section.data1.start=0x60，则：

1．section.data2.start=（ ）

2．section.data3.start=（ ）

3．执行mov ax,lba 指令后，寄存器AX 中的内容是多少？

4．执行mov ax,lbc 指令后，寄存器AX 中的内容是多少？

5．执行mov ax,lbd 指令后，寄存器AX 中的内容是多少？

```
        ......          ;其他指令
section data1 align=16 vstart=0
        lba db 0x55,0xf0
section data2 align=16 vstart=0
        lbb db 0x00,0x90
        lbc dw 0xf000
section data3 align=16
        lbd dw 0xfff0,0xfffc
```

8.2.2　用户程序头部

在上面，我们已经知道如何在用户程序中分段，也知道各种段定义子句对段的起始汇编地址和段内汇编地址的影响。现在，让我们结合本章中的实例来进一步加深认识。

浏览一下本章代码清单8-2，你会发现，本章的用户程序实际上定义了7 个段，分别是第7行定义的段header、第27 行定义的段code_1、第163 行定义的段code_2、第173 行定义的段data_1、第194 行定义的段data_2、第201 行定义的段stack 和第208 行定义的段trail。

一般来说，加载器和用户程序是在不同的时间、不同的地方，由不同的人或公司开发的。这就意味着，它们彼此并不了解对方的结构和功能。事实上，也不需要了解。

如图8-5 所示，它们彼此看对方都是一个黑盒子，并不了解对方是怎么编写的，是做什么的。但是，也不能完全是黑的，加载器必须了解一些必要的信息，虽然不是很多，但足以知道如何加载用户程序。

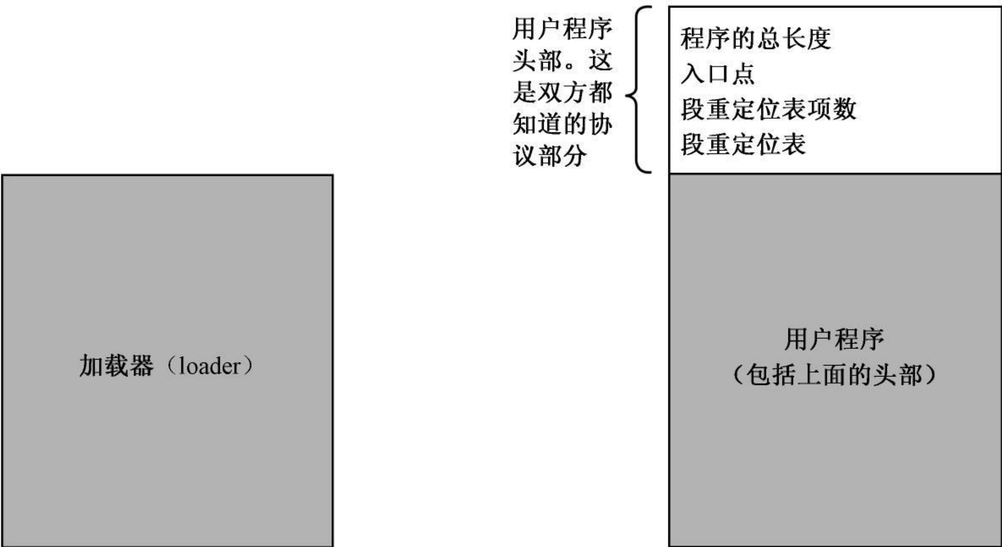


图8-5　加载器与用户程序之间的协议部分示意图

这就涉及加载器的编写者，以及用户程序的编写者，他们之间是怎么协商的。他们之间必须有一个协议，或者说协定，比如说，在用户程序内部的某个固定位置，包含一些基本的结构信息，每个用户程序都必须把自己的情况放在这里，而加载器也固定在这个位置读取。经验表明，把这个约定的地点放在用户程序的开头，对双方，特别是对加载器来说比较方便，这就是用户程序头部。

头部需要在源程序以一个段的形式出现。这就是代码清单8-2 的第7行：

```
SECTION header vstart=0
```

而且，因为它是“头部”，所以，该段当然必须是第一个被定义的段，且总是位于整个源程序的开头。

用户程序头部起码要包含以下信息。

① 用户程序的尺寸，即以字节为单位的大小。这对加载器来说是很重要的，加载器需要根据这一信息来决定读取多少个逻辑扇区（在本书中，所有程序在硬盘上所占用的逻辑扇区都是连续的）。

代码清单8-2 中第8 行，伪指令dd 用于声明和初始化一个双字，即一个32 位的数据。用户程序可能很大，16 位的长度不足以表示65535 以上的数值。

程序的长度取自程序中的一个标号“program_end”，这是允许的。在编译阶段，编译器将该标号所代表的汇编地址填写在这里。该标号位于整个源程序的最后，从属于段“trail”。由于该段并没有vstart 子句，所以，标号“program_end”所代表的汇编地址是从整个程序的开头计算的。换句话说，program_end 所代表的汇编地址，在数值上等于整个程序的长度。

双字在内存中的存放也是按低端序的。如图8-6 所示，低字保存在低地址，高字保存在高地址。同时，每个字又按低端字节序，低字节在低地址，高字节在高地址。

② 应用程序的入口点，包括段地址和偏移地址。加载器并不清楚用户程序的分段情况，更不知道第一条要执行的指令在用户程序中的位置。因此，必须在头部给出第一条指令的段地址和偏移地址，这就是所谓的应用程序入口点（Entry Point）。

理想情况下，当用户程序开始运行时，执行的第一条指令是其代码段内的第一条指令。换句话说，入口点位于其代码段内偏移地址为0 的地方。但是，情况并非总是如此。尤其是，很多程序并非只有一个代码段，比如本章源代码清单8-2 就包含了两个代码段。所以，需要在用户程序头部明确给出用户程序在刚开始运行时，第一条指令的位置，也就是第一条指令在用户程序代码段内的偏移地址。

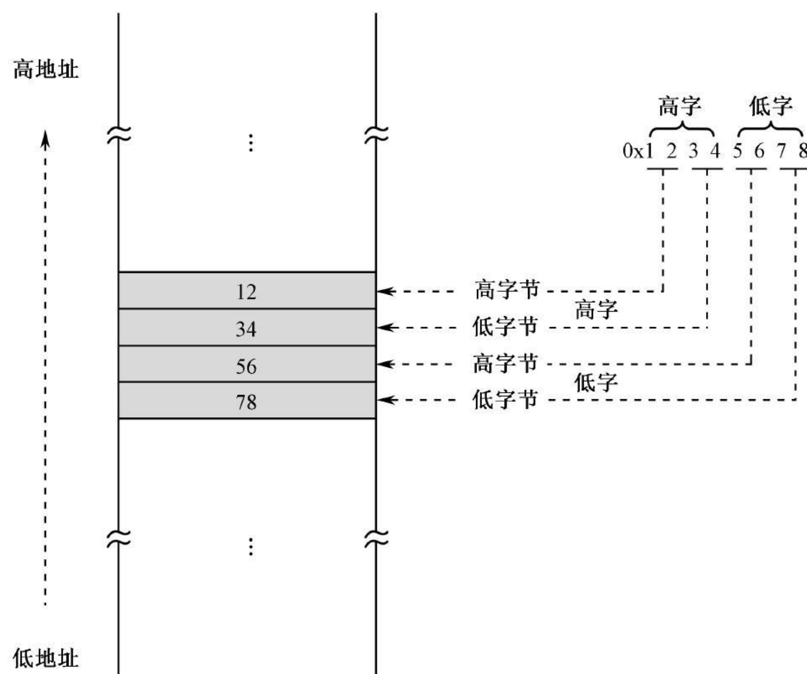


图8-6　双字数据在内存中的布局

代码清单8-2 第11、12 行，依次声明并初始化了入口点的偏移地址和段地址。偏移地址取自代码段code_1 中的标号“start”，段地址是用表达式section.code_1.start 得到的。

代码段code_1 是在代码清单8-2 的第27 行定义的：

```
SECTION code_1 align=16 vstart=0
```

显而易见的是，因为段定义中包含了“vstart=0”子句，故标号start 所代表的汇编地址是相对于当前代码段code_1 的起始位置，从0 开始计算的。

入口点的段地址是用伪指令dd 声明的，并初始化为汇编地址section.code_1.start，这是一个32 位的地址。不过，它仅仅是编译阶段确定的汇编地址，在用户程序加载到内存后，需要根据加载的实际位置重新计算（浮动）。

尽管在16 位的环境中，一个段最长为64KB，但它却可以起始于任何20 位的物理地址处。你不可能用16 位的单元保存20 位的地址，所以，只能保存为32 位的形式。

③ 段重定位表。用户程序可能包含不止一个段，比较大的程序可能会包含多个代码段和多个数据段。这些段如何使用，是用户程序自己的事，但前提是程序加载到内存后，每个段的地址必须重新确定一下。

段的重定位是加载器的工作，它需要知道每个段在用户程序内的位置，即它们分别位于用户程序内的多少字节处。为此，需要在用户程序头部建立一张段重定位表。

用户程序可以定义的段在数量上是不确定的，因此，段重定位表的大小，或者说表项数是不确定的。为此，代码清单8-2 第14 行，声明并初始化了段重定位表的项目数。因为段重定位表位于两个标号header_end 和code_1_segment 之间，而且每个表项占用4 字节，故实际的表项数为

```
(header_end - code_1_segment) / 4
```

这个值是在程序编译阶段计算的，先用两个标号所代表的汇编地址相减，再除以每个表项的长度4。

紧接着表项数的，是实际的段重定位表，每个表项用伪指令dd 声明并初始化为1 个双字。代码清单8-2 一共定义了5 个段，所以这里有5 个表项，依次计算段开始汇编地址的表达式并进行初始化。

8.3 加载程序（器）的工作流程

8.3.1 初始化和决定加载位置

从大的方面来说，加载器要加载一个用户程序，并使之开始执行，需要决定两件事。第一，看看内存中的什么地方是空闲的，即从哪个物理内存地址开始加载用户程序；第二，用户程序位于硬盘上的什么位置，它的起始逻辑扇区号是多少。如果你连它在哪里都不知道，怎么找得到它呢！

现在，让我们把目光转移到代码清单8-1，来看看加载器都做了哪些工作。

代码清单8-1 第6 行，加载器程序的一开始声明了一个常数（const）：

```
app_lba_start equ 100
```

常数是用伪指令equ 声明的，它的意思是“等于”。本语句的意思是，用标号app_lba_start 来代表数值100，今后，当我们要用到100 的时候，不这样写：

```
mov al,100
```

而是这样写：

```
mov al,app_lba_start
```

你可能会说，这样不是更麻烦吗？

不会的，实际上这很方便。用某些教材上的话说，程序中不该使用“不可思议的数”。想想看，如果在程序中的多个地方直接使用数值100，那么，以后要修改它们，把它们改成500，还得找到所有使用这个数值的位置，一一修改，万一漏掉一个呢？如果使用常量app_lba_start，则只需要重新把这个常数的声明语句改成下面的形式，并重新编译即可。

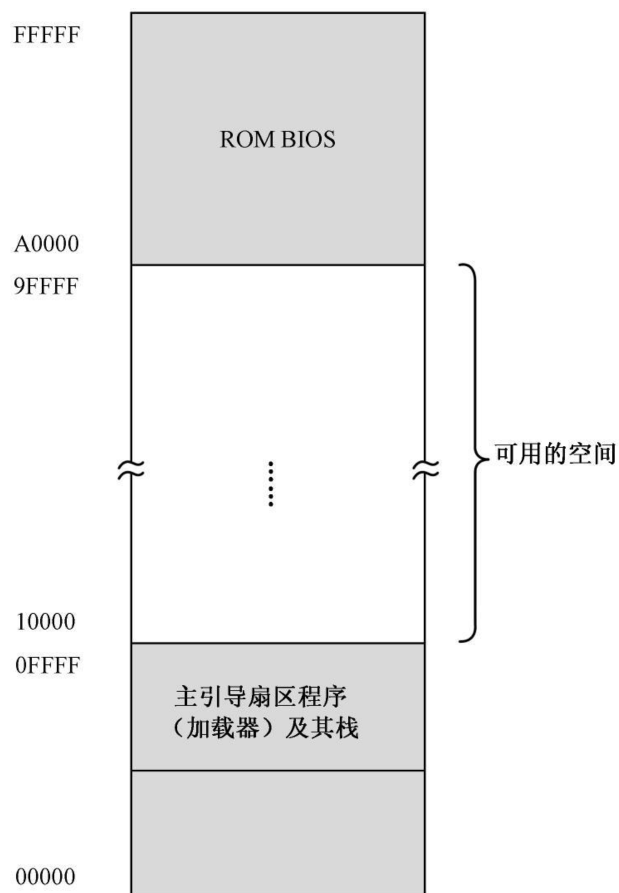


图8-7　可用于加载用户程序的空间范围

```
app_lba_start equ 500
```

常数的意思是在程序运行期间不变的数。和其他伪指令db、dw、dd不同，用equ 声明的数值不占用任何汇编地址，也不在运行时占用任何内存位置。它仅仅代表一个数值，就这么简单。

加载用户程序需要确定一个内存物理地址，这是在代码清单8-1 第151 行用伪指令dd 声明的，并初始化为0x10000 的。和前面一样，是用32 位的单元来容纳一个20 位的地址：

```
phy_base dd 0x10000
```

尽管我们用了一个好看的数0x10000，但你完全可以把用户程序加载到其他地方，只要它是空闲的。比如，可以将这个数值改成0x12340，唯一的要求是该地址的最低4 位必须是0，换句话说，加载的起始地址必须是16 字节对齐的，这样将来才能形成一个有效的段地址。

如图8-7 所示，物理地址0x0FFFF 以下，是加载器及其栈的势力范围；物理地址A0000 以上，是BIOS 和外围设备的势力范围，有很多传统的老式设备将自己的存储器和只读存储器映射到这个空间。

如此一来，可用的空间就位于0x10000～9FFFF，差不多500 多KB。事实上，如果将低端的内存空间合理安排一下，还可以腾出更多空间，但是没有必要，我们用不了多少。

8.3.2 准备加载用户程序

和以往不同，我们将主引导扇区程序定义成一个段。代码清单8-1 第9 行：

```
SECTION mbr align=16 vstart=0x7c00
```

整个程序只定义了这一个段，所以它略显多余。之所以这么说，是因为，即使你不定义这个段，编译器也会自动把整个程序看成一个段。

但是，因为该定义中有“vstart=0x7c00”子句，所以，它就不那么多余了。一旦有了该子句，段内所有元素的汇编地址都将从0x7c00 开始计算。否则，因为主引导程序的实际加载地址是0x0000:0x7c00，当我们引用一个标号时，还得手工加上那个落差0x7c00。

代码清单8-1 第12～14 行，用于初始化栈段寄存器SS 和栈指针SP。之后，栈的段地址是0x0000，段的长度是64KB，栈指针将在段内0xFFFF 和0x0000 之间变化。

代码清单8-1 第16、17 行，用于取得一个地址，用户程序将要从这个地址处开始加载。

该地址实际上是保存在标号phy_base 处的一个双字单元里。这是一个32 位的数，在16 位的处理器上，只能用两个寄存器存放。如图8-8 所示，32 位数内存中的存放是按低端序列的，高16位处在phy_base＋0x02 处，可以放在寄存器DX 中；低16 位处在phy_base 处，可以用寄存器AX存放。

这两条指令中都使用了段超越前缀“cs:”。这是允许的，意味着在访问内存单元时，使用CS的内容作为段基址。之所以没有使用DS 和ES，是因为它们另有安排。

另外注意，因为段寄存器CS 的内容是0x0000，而且主引导扇区是位于0x0000:0x7c00 处的，所以，理论上指令中的偏移地址应当是0x7c00＋phy_base。不过，因为我们定义段mbr 的时候，使用了“vstart=0x7c00”子句，故段内所有汇编地址都是在0x7c00 的基础上增加的，就不用再加上这个0x7c00 了，直接是

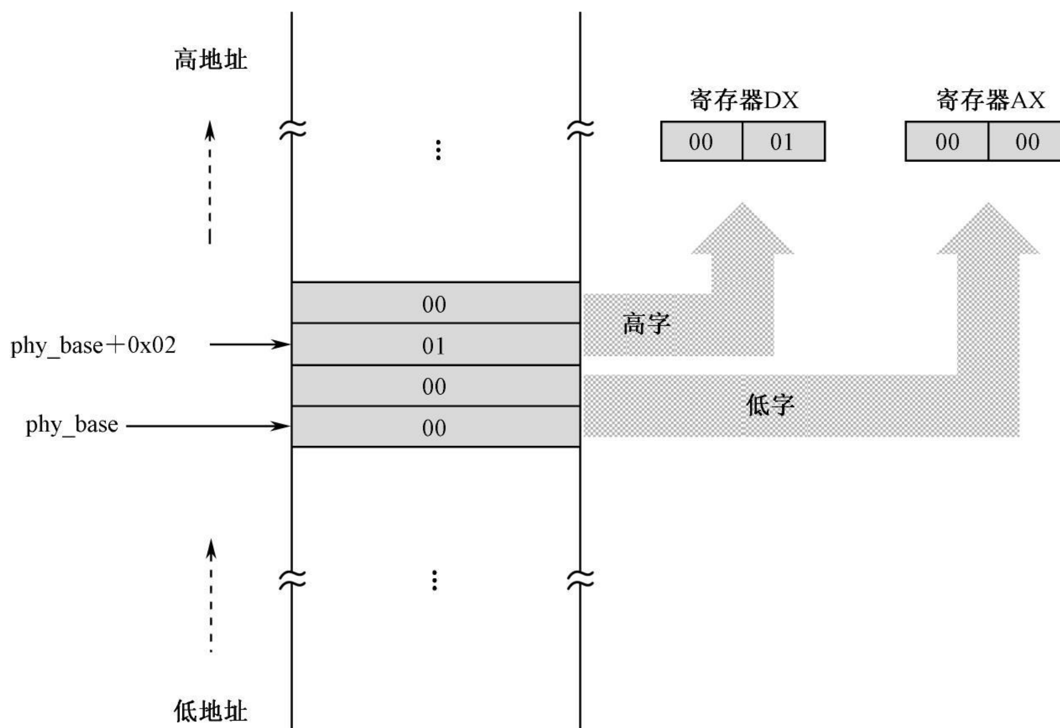


图8-8　获取用于加载用户程序的物理地址

```
mov ax,[cs:phy_base]
mov dx,[cs:phy_base+0x02]
```

紧接着，代码清单8-1 第18～21 行，用于将该物理地址变成16 位的段地址，并传送到DS 和ES 寄存器。因为该物理地址是16 字节对齐的，直接右移4 位即可。实际上，右移4 位相当于除以16（0x10），所以程序中的做法将这个32 位物理地址（DX:AX）除以16（在寄存器BX 中），寄存器AX 中的商就是得到的段地址（在本程序中是0x1000）。

8.3.3　外围设备及其接口

加载器的下一个工作是从硬盘读取用户程序，说白了就是访问其他硬件。和处理器打交道的硬件很多，不单单是硬盘，还有显示器、网络

设备、扬声器（喇叭）和话筒（麦克风）、键盘、鼠标等。有时候，根据应用的场合，还会接一些你不认识和没见过的东西。

所有这些和计算机主机连接的设备，都围绕在主机周围，争着跟计算机说话，叫做外围设备（Peripheral Equipment）。一般来说，我们把这些设备分成两种，一种是输入设备，比如键盘、鼠标、麦克风、摄像头等；另一种是输出设备，比如显示器、打印机、扬声器等。输入设备和输出设备统称输入输出（Input/Output，I/O）设备。

每一种设备都有自己的怪脾气，都有和别的设备不一样的工作方式。比如，扬声器需要的是模拟信号，每个扬声器需要两根线，用的插头也是无线电行业里的标准，话筒也是如此；老式键盘只用一根线向主机传送按键的ASCII 码，而且一直采用PS/2 标准；新式的USB 键盘尽管也使用串行方式工作，但信号却和老式键盘完全不同。至于网络设施，现在流行的是里面有8 根线芯的五类双绞线，里面的信号也有专门的标准。

一句话，不同的设备，有不同的连线数量，线里面传送的信号也不一样，而且各自的插头和插孔也千差万别，这该如何让处理器跟它打交道？

话虽这么说，但这些东西不让处理器访问和控制却不行。很明显，这里需要一些信号转换器和变速齿轮，这就是I/O 接口。举几个例子，麦克风和扬声器需要一个I/O 接口，即声卡，才能与处理器沟通；显示器也需要一个I/O 接口，即显卡，才能与处理器沟通；USB 键盘同样需要一个I/O 接口，即USB 接口，才能与处理器沟通。很显然，不同的外围设备，都有各自不同的I/O 接口。

I/O 接口可以是一个电路板，也可能是一块小芯片，这取决于它有多复杂。无论如何，它是一个典型的变换器，或者说是一个翻译器，在一边，它按处理器的信号规程工作，负责把处理器的信号转换成外围设备能接受的另一种信号；在另一边，它也做同样的工作，把外围设备的信号变换成处理器可以接受的形式。

这还没完，后面还有两个麻烦的问题。

① 不可能将所有的I/O 接口直接和处理器相连，设备那么多，还有些设备现在没有发明出来，将来一定会有。你怎么办？

② 每个设备的I/O 接口都抢着和处理器说话，不发生冲突都难。你怎么办？

对第1 个问题的解答是采用总线技术。总线可以认为是一排电线，所有的外围设备，包括处理器，都连接到这排电线上。但是，每个连接到这排电线上的器件都必须有拥有电子开关，以使它们随时能够同这排电线连接，或者从这排电线上断开（脱离）。这就好比是公共车道，当路面上有车时，你就必须退避一下，不能硬冲上去。因此，这排公共电线就称为总线（Bus）。

对第2 个问题的解答是使用输入输出控制设备集中器（I/O Controller Hub，ICH）芯片，该芯片的作用是连接不同的总线，并协调各个I/O 接口对处理器的访问。在个人计算机上，这块芯片就是所谓的南桥。

如图8-9 所示，处理器通过局部总线连接到ICH 内部的处理接口电路。然后，在ICH 内部，又通过总线与各个I/O 接口相连。

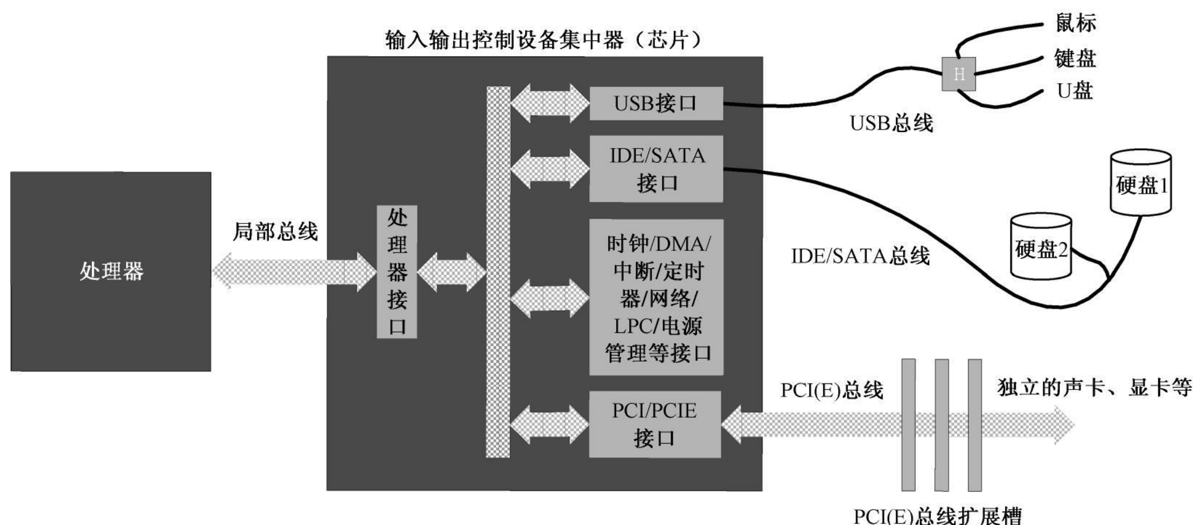


图8-9　计算机内部总线系统示意图

在ICH 内部，集成了一些常规的外围设备接口，如USB、PATA（IDE）、SATA、老式总线接口（LPC）、时钟等，这些东西对计算机来说必不可少，故直接集成在ICH 内，我们后面还会详细介绍它们的功能。

除了这些常用的、必不可少的设备之外，有些设备你可能暂时用不上，也有些设备还没有发明出来，但迟早有可能连在计算机上。不管是什么设备，都必须通过它自己的I/O 接口电路同ICH 相连。为了方便，最

好是在主板上做一些插槽，同时，每个设备的I/O 接口电路都设计成插卡。这样，想接上该设备时，就把它的I/O 接口卡插上，不需要时，随时拔下。

为了实现这个目的，或者说为了支持更多的设备，ICH 还提供了对PCI 或者PCI Express 总线的支持，该总线向外延伸，连接着主板上的若干个扩展槽，就是刚才说的插槽。举个实例，如果你想连接显示器，那么就要先插入显卡，然后再把显示器接到显卡上。

除了局部总线和PCI Express 总线，每个I/O 接口卡可能连接不止一个设备。比如USB 接口，就有可能连接一大堆东西：键盘、鼠标、U 盘等。因为同类型的设备较多，也涉及线路复用和仲裁的问题，故它们也有自己的总线体系，称为通信总线或者设备总线。比如图8-9 所示的USB 总线和SATA 总线。

当处理器想同某个设备说话时，ICH 会接到通知。然后，它负责提供相应的传输通道和其他辅助支持，并命令所有其他无关设备闭嘴。同样，当某个设备要跟处理器说话，情况也是一样。

8.3.4　I/O 端口和端口访问

外围设备和处理器之间的通信是通过相应的I/O 接口进行的。当然，这么说太过于笼统，所以必须具体到细节上来讲这件事。

具体地说，处理器是通过端口（Port）来和外围设备打交道的。本质上，端口就是一些寄存器，类似于处理器内部的寄存器。不同之处仅仅在于，这些叫做端口的寄存器位于I/O 接口电路中。

端口是处理器和外围设备通过I/O 接口交流的窗口，每一个I/O 接口都可能拥有好几个端口，分别用于不同的目的。比如，连接硬盘的PATA/SATA 接口就有几个端口，分别是命令端口（当向该端口写入0x20 时，表明是从硬盘读数据；写入0x30 时，表明是向硬盘写数据）、状态端口（处理器根据这个端口的数据来判断硬盘工作是否正常，操作是否成功，发生了哪种错误）、参数端口（处理器通过这些端口告诉硬盘读写的扇区数量，以及起始的逻辑扇区号）和数据端口（通过这个端口连续地取得要读出的数据，或者通过这个端口连续地发送要写入硬盘的数据）。

端口只不过是位于I/O 接口上的寄存器，所以，每个端口有自己的数据宽度。在早期的系统中，端口可以是8 位的，也可以是16 位的，现在有些端口会是32 位的。到底是8 位还是16 位，这是设备和I/O 接口制造者的自由。比如，PATA/STAT 接口中的数据端口就是16 位的，这有助于加快数据传输速率，提高传输效率。

端口在不同的计算机系统中有着不同的实现方式。在一些计算机系统中，端口号是映射到内存地址空间的。比如，0x00000～0xE0000 是真实的物理内存地址，而0xE0001～0xFFFFF 是从很多I/O 接口那里映射过来的，当访问这部分地址时，实际上是在访问I/O 接口。

而在另一些计算机系统中，端口是独立编址的，不和内存发生关系。如图8-10 所示，在这种计算机中，处理器的地址线既连接内存，也连接每一个I/O 接口。但是，处理器还有一个特殊的引脚M/IO#，在这里，“#”表示低电平有效。也就是说，当处理器访问内存时，它会让M/IO#引脚呈高电平，这里，和内存相关的电路就会打开；相反，如果处理器访问I/O 端口，那么M/IO#引脚呈低平，内存电路被禁止。与此同时，处理器发出的地址和M/IO#信号一起用于打个某个I/O 接口，如果该I/O 接口分配的端口号与处理器地址相吻合的话。

Intel 处理器，早期是独立编址的，现在既有内存映射的，也有独立编址的。在本章中，我们只讲独立编址的端口。

所有端口都是统一编号的，比如0x0001、0x0002、0x0003、…。每个I/O 接口电路都分配了若干个端口，比如，I/O 接口A 有3 个端口，端口号分别是0x0021～0x0023；I/O 接口B 需要5 个端口，端口号分别是0x0303～0x0307。

一个现实的例子是个人计算机中的PATA/SATA 接口（图8-9），每个PATA 和SATA 接口分配了8个端口。但是，ICH 芯片内部通常集成了两个PATA/SATA 接口，分别是主硬盘接口和副硬盘接口。这样一来，主硬盘接口分配的端口号是0x1f0～0x1f7，副硬盘接口分配的端口号是0x170～0x177。

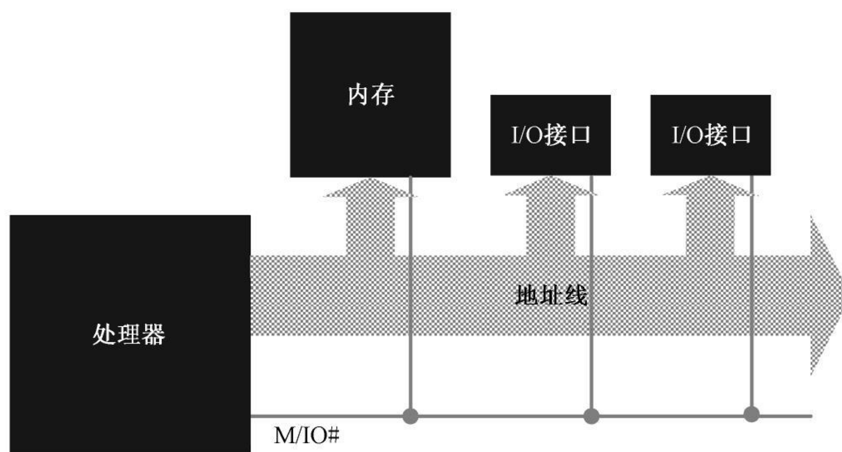


图8-10　端口的访问和M/IO#引脚

在Intel 的系统中，只允许65536（十进制数）个端口存在，端口号从0 到65535（0x0000～0xffff）。因为是独立编址，所以，端口的访问不能使用类似于mov 这样的指令，取而代之的是in 和out 指令。

in 指令是从端口读，它的一般形式是

```
in al,dx
```

或者

```
in ax,dx
```

这就是说，in 指令的目的操作数必须是寄存器AL 或者AX，当访问8 位的端口时，使用寄存器AL；访问16 位的端口时，使用AX。in 指令的源操作数应当是寄存器DX。

in al,dx 的机器指令码是0xEC，in ax,dx 的机器指令码是0xED，都是一字节的。之所以如此简短，是因为in 指令不允许使用别的通用寄存器，也不允许使用内存单元作为操作数。

也许是为了方便，in 指令还有两字节的形式。此时，前一字节是操作码0xE4 或者0xE5，分别用于指示8 位或者16 位端口访问；后一字节是立即数，指示端口号。

因此，机器指令 E4 F0 就相当于汇编语言指令

```
in al,0xf0
```

而机器指令E5 03 就相当于汇编语言指令

```
in ax,0x03
```

很显然，因为这种指令形式的操作数部分只允许一字节，故只能访问0～255（0x00～0xff）号端口，不允许访问大于255 的端口号。所以，下面的汇编语言指令就是非法的：

```
in ax,0x5fd
```

in 指令不影响任何标志位。

相应地，如果要通过端口向外围设备发送数据，则必须通过out 指令。

out 指令正好和in 指令相反，目的操作数可以是8 位立即数或者寄存器DX，源操作数必须是寄存器AL 或者AX。下面是一些例子：

```
out 0x37,al            ;写 0x37 号端口（这是一个 8 位端口）
out 0xf5,ax            ;写 0xf5 号端口（这是一个 16 位端口）
out dx,al              ;这是一个 8 位端口，端口号在寄存器 DX 中
out dx,ax              ;这是一个 16 位端口，端口号在寄存器 DX 中
```

和in 指令一样，out 指令不影响任何标志位。

8.3.5　通过硬盘控制器端口读扇区数据

现在，让我们来看看硬盘。

硬盘读写的基本单位是扇区。就是说，要读就至少读一个扇区，要写就至少写一个扇区，不可能仅读写一个扇区中的几个字节。这样一来，就使得主机和硬盘之间的数据交换是成块的，所以硬盘是典型的块设备。

从硬盘读写数据，最经典的方式是向硬盘控制器分别发送磁头号、柱面号和扇区号（扇区在某个柱面上的编号），这称为CHS 模式。这种方法最原始，最自然，也最容易理解。

实际上，在很多时候，我们并不关心扇区的物理位置，所以希望所有的扇区都能统一编址。这就是逻辑扇区，它把硬盘上所有可用的扇区

都一一从0 编号，而不管它位于哪个盘面，也不管它属于哪个柱面。

关于硬盘和逻辑扇区的知识前面已经有所介绍，这里不再赘述。最早的逻辑扇区编址方法是LBA28，使用28 个比特来表示逻辑扇区号，从逻辑扇区0x0000000 到0xFFFFFFF，共可以表示228＝268435456 个扇区。每个扇区有512 字节，所以LBA28 可以管理128 GB 的硬盘。

硬盘技术发展得非常快，最新的硬盘已经达到几百个吉字节的容量，LBA28 已经落后了。在这种情况下，业界又共同推出了LBA48，采用48 个比特来表示逻辑扇区号。如此一来，就可以管理131072 TB 的硬盘容量了。

```
1GB = 1024MB
```

```
1TB = 1024GB
```

在本章中，我们将采用LBA28 来访问硬盘。

前面说过，个人计算机上的主硬盘控制器被分配了8 位端口，端口号从0x1f0 到0x1f7。假设现在要从硬盘上读逻辑扇区，那么，整个过程如下。

第1 步，设置要读取的扇区数量。这个数值要写入0x1f2 端口。这是个8 位端口，因此每次只能读写255 个扇区：

```
mov dx,0x1f2
mov al,0x01                    ;1 个扇区
out dx,al
```

注意，如果写入的值为0，则表示要读取256 个扇区。每读一个扇区，这个数值就减一。因此，如果在读写过程中发生错误，该端口包含着尚未读取的扇区数。

第2 步，设置起始LBA 扇区号。扇区的读写是连续的，因此只需要给出第一个扇区的编号就可以了。28 位的扇区号太长，需要将其分成4 段，分别写入端口0x1f3、0x1f4、0x1f5 和0x1f6 号端口。其中，0x1f3 号端口存放的是0～7 位；0x1f4 号端口存放的是8～15 位；0x1f5 号端口存放的是16～23 位，最后4 位在0x1f6 号端口。假定我们要读写的起始逻辑扇区号为0x02，可编写代码如下：

```
mov dx,0x1f3
mov al,0x02
out dx,al        ;LBA 地址 7～0
inc dx           ;0x1f4
mov al,0x00
out dx,al        ;LBA 地址 15～8
inc dx           ;0x1f5
out dx,al        ;LBA 地址 23～16
inc dx           ;0x1f6
mov al,0xe0      ;LBA 模式，主硬盘，以及 LBA 地址 27～24
out dx,al
```

注意以上代码的最后4 行，在现行的体系下，每个PATA/SATA 接口允许挂接两块硬盘，分别是主盘（Master）和从盘（Slave）。如图8-11 所示，0x1f6 端口的低4 位用于存放逻辑扇区号的24～27位，第4 位用于指示硬盘号，0 表示主盘，1 表示从盘。高3 位是“111”，表示LBA 模式。

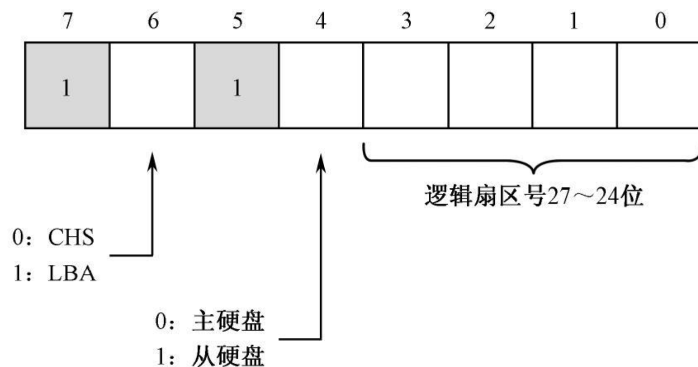


图8-11　端口1f6 各位的含义

第3 步，向端口0x1f7 写入0x20，请求硬盘读。这也是一个8 位端口：

```
mov dx,0x1f7
```

```
mov al,0x20             ;读命令
out dx,al
```

第4 步，等待读写操作完成。端口0x1f7 既是命令端口，又是状态端口。在通过这个端口发送读写命令之后，硬盘就忙乎开了。如图8-12 所示，在它内部操作期间，它将0x1f7 端口的第7位置“1”，表明自己很忙。一旦硬盘系统准备就绪，它再将此位清零，说明自己已经忙完了，同时

将第3 位置“1”，意思是准备好了，请求主机发送或者接收数据（图8-12）。完成这一步的典型代码如下:

```
        mov dx,0x1f7
    .waits:
        in al,dx
        and al,0x88
        cmp al,0x08
        jnz .waits              ;不忙，且硬盘已准备好数据传输
```

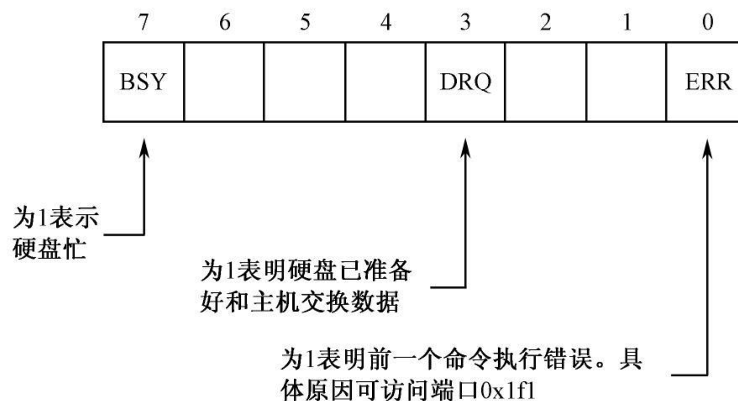


图8-12　端口0x1f7 部分状态位的含义

来看看指令and al,0x88。0x88 的二进制形式是10001000，这意味着我们想用这条指令保留住寄存器AL 中的第7 位和第3 位，其他无关的位都清零。此时，如果寄存器AL 中的二进制数是00001000（0x08），那就说明可以退出等待状态，继续往下操作，否则继续等待。

第5 步，连续取出数据。0x1f0 是硬盘接口的数据端口，而且还是一个16 位端口。一旦硬盘控制器空闲，且准备就绪，就可以连续从这个端口写入或者读取数据。下面的代码假定是从硬盘读一个扇区（512 字节，或者256 字节），读取的数据存放到由段寄存器DS 指定的数据段，偏移地址由寄存器BX 指定：

```
        mov cx,256              ;总共要读取的字数
        mov dx,0x1f0
    .readw:
        in ax,dx
        mov [bx],ax
        add bx,2
        loop .readw
```

最后，0x1f1 端口是错误寄存器，包含硬盘驱动器最后一次执行命令后的状态（错误原因）。

8.3.6　过程调用

读写硬盘是经常要做的事，尤其对于操作系统来说。即使是在本章的程序中，也多次发生。如果每次读写硬盘都按上面的5 个步骤写一堆代码，程序势必很大，也会令人烦恼。

好在处理器支持一种叫过程调用的指令执行机制。过程（Procedure）又叫例程，或者子程序、子过程、子例程（Sub-routine），不管怎么称呼，实质都一样，都是一段普通的代码。处理器可以用过程调用指令转移到这段代码执行，在遇到过程返回指令时重新返回到调用处的下一条指令接着执行。

如图8-13 所示，这是过程和过程调用的示意图。下面结合本章代码清单来具体说明。

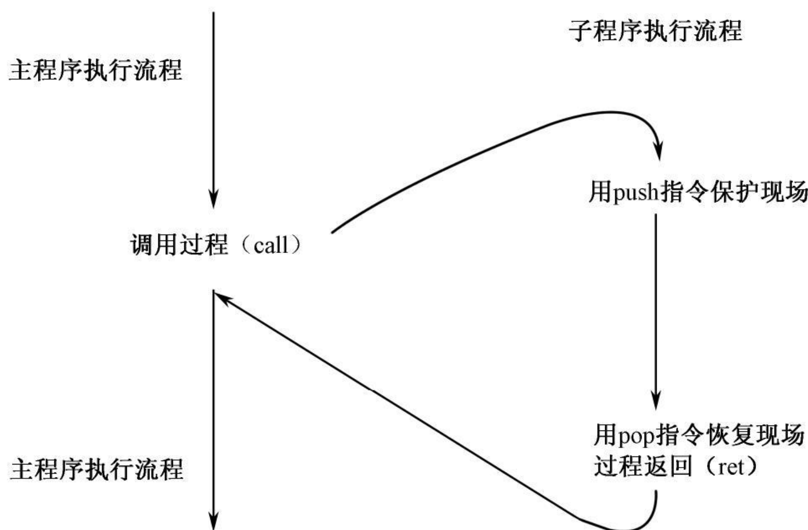


图8-13　过程和过程调用示意图

在8.3.1 节里，我们已经定义了常量app_lba_start，它代表的值是100，也就是用户程序在硬盘上的起始逻辑扇区号。现在，代码清单8-1的第24～27 行用于从硬盘上读取这个扇区的内容。这很好理解，因为不知道用户程序到底有多大，到底占用了多少个扇区，所以，可以先读它的第一个扇区。该扇区包含了用户程序的头部，而用户程序头部又包含

了该程序的大小、入口点和段重定位表。所以，通过分析头部，就知道接着还要再读多少个扇区才能完全加载用户程序。

因为要多次读取硬盘，而每次的步骤又都差不多，所以，我们精心设计了一段通用的代码，它从代码清单8-1 的第79 行开始，一直到第131 行结束，这就是我们所说的过程。

要调用过程，需要该过程的地址。一般来说，过程的第一条指令需要一个标号，以方便引用该过程。所以，代码清单8-1 第79 行是一个标号“read_hard_disk_0”，意思是读（第一个硬盘控制器的）主盘，当然，什么意思并不重要。

编写过程的好处是只用编写一次，以后只需要“调用”即可。所以，代码的灵活性和通用性尤其重要。具体到这里，就是每次读硬盘时的起始逻辑扇区号和数据保存位置都不相同，这就涉及所谓的参数传递。

参数传递最简单的办法是通过寄存器。在这里，主程序把起始逻辑扇区号的高16 位存放在寄存器DI 中（只有低12 位是有效的，高4 位必须保证为“0”），低16 位存放在寄存器SI 中（没办法，16 位的处理器无法直接处理28 位的数据）；并约定将读出来的数据存放到由段寄存器DS 指向的数据段中，起始偏移地址在寄存器BX 中。

在调用过程前，程序会用到一些寄存器，在过程返回之后，可能还要继续使用。为了不失连续性，在过程的开头，应当将本过程要用到（内容肯定会被破坏）的寄存器临时压栈，并在返回到调用点之前出栈恢复。代码清单8-1 的第82～85 行，用于将过程中用到的寄存器入栈保存。

后面的指令都很好理解，第87～89 行，是向0x1f2 端口写入要读取的扇区数。显而易见，每次读的扇区数是1 个。

第91～101 行，用于向硬盘接口写入起始逻辑扇区号的低24 位。低16 位在寄存器SI 中，高12 位在寄存器DI 中，需要不停地倒换到寄存器AL 中，以方便端口写入。

第105 行，程序执行到这里时，寄存器AH 的低4 位是起始逻辑扇区号的27～24 位，高4 位是全“0”；寄存器AL 中是0xe0。执行or 指令后，将会在寄存器AL 中得到它们的组合值，高4位是0xe，低4 位是逻辑扇区号的27～24 位。

第118～124 行，用于反复从硬盘接口那里取得512 字节的数据，并传送到段寄存器DS 所指向的数据区中。每传送一个字，BX 的值就增2，以指向下一个偏移位置。

第126～129 行，用于把调用过程前各个寄存器的内容从栈中恢复。

最后，因为处理器是没有大脑的，所以需要一个明确的指令ret 促使它离开过程，从哪里来回哪里去，这条指令稍后就会讲到。

有关过程的情况就是这些，下面回到前面，看看过程调用是如何发生的。

代码清单8-1 第24、25 行，用于指定用户程序在硬盘上的起始逻辑扇区号。我们定义的过程要求用DI:SI 来提供这个扇区号，既然它是常数100，很小的数值，可以直接传送到寄存器SI，并将DI 清零即可。

第26 行用于指定存放数据的内存地址。前面几条指令已经将段寄存器DS 设置好了，现在只需要将寄存器BX 清零，以指向该段内偏移地址为0 的地方，这就是当前指令要做的事。

一切都准备好了，第27 行，开始调用过程read_hard_disk_0。以后，我们将把过程所在的标号做为过程的名字，即过程名。

调用过程的指令是“call”。8086 处理器支持四种调用方式。

第一种是16 位相对近调用 。近调用的意思是被调用的目标过程位于当前代码段内，而非另一个不同的代码段，所以只需要得到偏移地址即可。

16 位相对近调用是三字节指令，操作码为0xE8，后跟16 位的操作数，因为是相对调用，故该操作数是当前call 指令相对于目标过程的偏移量。计算过程如下：用目标过程的汇编地址减去当前call 指令的汇编地址，再减去当前call 指令以字节为单位的长度（3），保留16 位的结果。举个例子：

```
        call near proc_1
```

近调用的特征是在指令中使用关键字“near”。“proc_1”是程序中的一个标号。在编译阶段，编译器用标号proc_1 处的汇编地址减去本指令的汇编地址，再减去3，作为机器指令的操作数。

关键字“near”不是必需的，如果call 指令中没有提供任何关键字，则编译器认为该指令是近调用。因此，上面的指令与这条指令等效：

```
call proc_1
```

因为16 位相对近调用的操作数是两个汇编地址相减的相对量，所以，如果被调用过程在当前指令的前方，也就是说，论汇编地址，它比call 指令的要大，那么该相对量是一个正数；反之，就是一个负数。所以，它的机器指令操作数是一个16 位的有符号数。换句话说，被调用过程的首地址必须位于距离当前call 指令－32768～32767 字节的地方。

在指令执行阶段，处理器看到操作码0xE8，就知道它应当调用一个过程。于是，它用指令指针寄存器IP 的当前内容加上指令中的操作数，再加上3，得到一个新的偏移地址。接着，将IP 的原有内容压入栈。最后，用刚才计算出的偏移地址取代IP 原有的内容。这直接导致处理器的执行流转移到目标位置处。

再看一个例子：

```
call 0x0500
```

很多人认为0x0500 会原封不动地出现在该指令编译后的机器码中，我相信这只是他们一时糊涂。在call 指令后跟一个标号，和跟一个数值没有什么不同。标号是数值的等价形式，是代表标号处的汇编地址。在指令编译阶段，它首先会被转化成数值。

所以，你在call 指令后跟一个数值，只是帮了编译器的忙，帮它省了一个转化步骤，它依然会用这个数值减去当前指令的汇编地址，来得到一个偏移量。

第二种是16 位间接绝对近调用 。这种调用也是近调用，只能调用当前代码段内的过程，指令中的操作数不是偏移量，而是被调用过程的真实偏移地址，故称为绝对地址。不过，这个偏移地址不是直接出现在指令中，而是由16 位的通用寄存器或者16 位的内存单元间接给出。比如：

```
call cx                    ;目标地址在 CX 中。省略了关键字“near”，下同
call [0x3000]              ;要先访问内存才能取得目标偏移地址
call [bx]                  ;要先访问内存才能取得目标偏移地址
call [bx+si+0x02]          ;要先访问内存才能取得目标偏移地址
```

以上，第一条指令的机器码为FF D1，被调用过程的偏移地址位于寄存器CX 内，在指令执行的时候由处理器从该寄存器取得，并直接取代指令指针寄存器IP 原有的内容。

第二条指令的机器码为FF 16 00 30。当这条指令执行时，处理器访问数据段（使用段寄存器DS），从偏移地址0x3000 处取得一个字，作为目标过程的真实偏移地址，并用它取代指令指针寄存器IP 原有的内容。

后面两条指令没什么好说的，只是寻址方式不同而已。

间接绝对近调用指令在执行时，处理器首先按以上的方法计算被调用过程的偏移地址，然后将指令指针寄存器IP 的当前值压栈，最后用计算出来的偏移地址取代寄存器IP 原有的内容。

由于间接绝对近调用的机器指令操作数是16 位的绝对地址，因此，它可以调用当前代码段任何位置处的过程。

第三种是16 位直接绝对远调用 。这种调用属于段间调用，即调用另一个代码段内的过程，所以称为远调用（far call）。很容易想到，远调用既需要被调用过程所在的段地址，也需要该过程在段内的偏移地址。

“16 位”是针对偏移地址来说的，而不是限定段地址，尽管段地址事实上也是16 位的；“直接”的意思是，段地址和偏移地址**直接** 在call 指令中给出了。当然，这里的地址也是绝对地址。比如：

```
call 0x2000:0x0030
```

这条指令编译后的机器码为9A 30 00 00 20，0x9A 是操作码，后面跟着的两个字分别是偏移地址和段地址，按规定，偏移地址在前，段地址在后。

处理器在执行时，首先将代码段寄存器CS 的当前内容压栈，接着再把指令指针寄存器IP 的当前内容压栈。紧接着，用指令中给出的段地址

代替CS 原有的内容，用指令中给出的偏移地址代替IP 原有的内容。这直接导致处理器从新的位置开始执行。

处理器是没有脑子的。如果被调用过程位于当前代码段内，而你又用这种指令格式来调用它，那么，处理器也会不折不扣地从当前代码段“转移”到当前代码段。

第四种是16 位间接绝对远调用 。这也属于段间调用，被调用过程位于另一个代码段内，而且，被调用过程所在的段地址和偏移地址是**间接** 给出的。还有，这里的“16 位”同样是用来限定偏移地址的。下面是这种调用方式的几个例子：

```
call far [0x2000]
call far [proc_1]
call far [bx]
call far [bx+si]
```

间接远调用必须使用关键字“far”，这一点务必牢记。

因为是远调用，也就是段间调用，所以，必须给出被调用过程的段地址和偏移地址。但是，段地址和偏移地址在内存中的其他位置，指令中仅仅给出的是该位置的偏移地址，需要处理器在执行指令的时候自行按图索骥，找到它们。

以上，前两条指令是等效的，不同之处仅仅在于，第一条指令直接给出的是数值，而第二条指令用的是标号。但这无关紧要，在编译后，标号也会变成数值。

为了进一步说清间接远调用是怎么发生的，下面是一个实例。

假如在数据段内声明了标号proc_1 并初始化了两个字：

```
proc_1 dw 0x0102,0x2000
```

这两个字分别是某个过程的段地址和偏移地址。按处理器的要求，偏移地址在前，段地址在后。也就是说，0x0102 是偏移地址； 0x2000 是段地址。

那么，为了调用该过程，可以在代码段内使用这条指令：

```
call far [proc_1]
```

当这条指令执行时，处理器访问由段寄存器DS 指向的数据段，从指令中指定的偏移地址（由标号proc_1 提供）处取得两个字（分别是段地址0x2000 和偏移地址0x0102）；接着，将代码段寄存器CS 和指令指针寄存器IP 的当前内容分别压栈；最后，用刚才取得的段地址和偏移地址分别取代CS 和IP 的原值。

至于后面的两条指令call far [bx]和call far [bx+si]，仅仅是寻址方式上有所区别，指令执行过程大体上是一样的。

接着回到代码清单8-1 第27 行，很明显，

```
        call read hard disk 0
```

就是我们刚刚讲的16 位相对近调用，编译后的机器指令操作数是一个相对偏移量。由于这是段内调用，处理器执行这条指令时，用指令指针寄存器IP 的内容加上指令中的偏移量，以及当前指令的长度，算出被调用过程的绝对偏移地址。接着，将IP 的现行值压栈。最后，用刚刚计算出的偏移地址替代IP 的当前内容。

过程read_hard_disk_0 的功能和工作流程前面已经讲过了，不再赘述。这里只关心一个最重要的问题，那就是过程返回。

“过程”就是例行公事，可以随时根据需要调用，但过程执行完了呢，还得返回到调用点继续执行下一条指令，这称为过程返回（Procedure Return）。

处理器是个大笨蛋，你不提醒它，它就一直稀里糊涂地闷头工作。幸好，处理器的发明者们设计了返回指令ret 和retf。

ret 和retf 经常用做call 和call far 的配对指令。ret 是近返回指令，当它执行时，处理器只做一件事，那就是从栈中弹出一个字到指令指针寄存器IP 中。

retf 是远返回指令（return far），它的工作稍微复杂一点点。当它执行时，处理器分别从栈中弹出两个字到指令指针寄存器IP 和代码段寄存器CS 中。

如图8-14 所示，在call read_hard_disk_0 执行前，栈指针位于箭头①所指示的位置；call 指令执行后，由于压入了IP 的内容，故栈指针移动到箭头②所指示的位置处；进入过程后，出于保护现场的目的，压入

了4 个通用寄存器AX、BX、CX、DX，此时，栈指针继续向低地址方向推进到箭头③所指示的位置。

在过程的最后，是恢复现场，连续反序弹出4 个通用寄存器的内容。此时，栈指针又回到刚进入过程内部时的位置，即箭头②处。最后，ret 指令执行时，由于处理器自动弹出一个字到IP，故，过程返回后的瞬间，栈指针仍旧回到过程调用前，即箭头①所指示的位置。

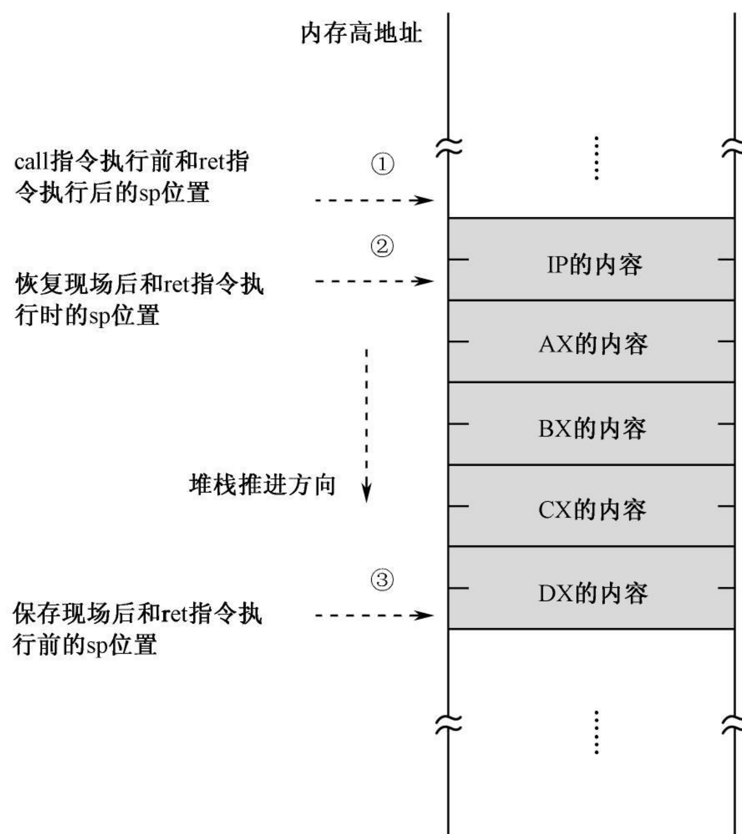


图8-14　过程调用前后的栈变化

需要说明的是，尽管call 指令通常需要ret/retf 和它配对，遥相呼应，但ret/retf 指令却并不依赖于call 指令，这一点你马上就会看到。

call 指令在执行过程调用时不影响任何标志位，ret/retf 指令对标志位也没有任何影响。

检测点8.2

按题目的要求写出相应的指令：

1．调用当前段内标号label_proc 处的过程；

2．调用当前段内的过程，过程的偏移地址在寄存器BX 中；

3．调用当前段内的过程，过程的偏移地址保存在当前数据段内由寄存器BX 所指向的内存单元中；

4．调用过程，过程的段地址为0xf000，偏移地址为0x0002；

5．调用过程，过程的段地址和偏移地址存放在当前数据段内偏移地址为0x80 的地方，低字是过程的偏移地址，高字为过程的段地址；

6．调用过程，过程的段地址和偏移地址存放在当前数据段内，低字为过程的偏移地址，高字为过程的段地址，这两个字在当前数据段内的偏移地址可以用BX+DI+0x08 得到。

8.3.7 加载用户程序

第一次读硬盘将得到用户程序最开始的512 字节，这512 字节包括最开始的用户程序头部，以及一部分实际的指令和数据。

为了将用户程序全部读入内存，需要知道它的大小，然后再进一步转换成它所用的扇区数。如图8-15 所示，用户程序最开始的双字，就是整个程序的大小。

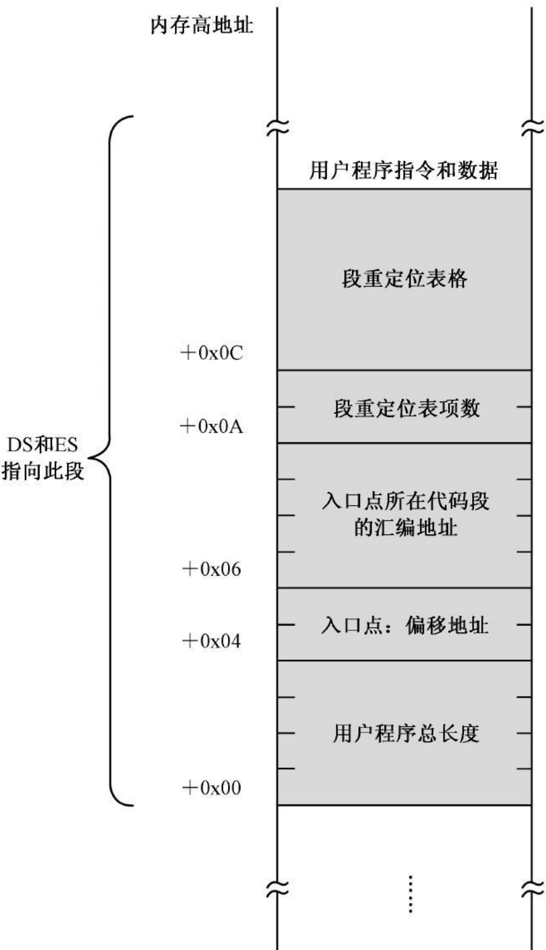


图8-15 用户程序头部结构示意图

为此，代码清单8-1 第30、31 行，分别将该数值的高16 位和低16 位传送到寄存器DX 和AX。第32 行，因为每扇区有512 字节，故将512 传送到BX 寄存器，并在第33 行用它来做除法运算。

在凑巧的情况下，用户程序的大小正好是512 的整数倍，做完除法后，在寄存器AX 中是用户程序实际占用的扇区数。但是，绝大多数情况下，这个除法会有余数。有余数意味着，最后一个扇区因为没有填满而落下了，没有纳入总扇区数。

关于这个问题，我们稍微解释一下。硬盘的读写是以扇区为单位的，如果要写入513 字节，那么，它将只能填满一个扇区，还剩一字节。硬盘不管这些，它每次总是说：“来，给我512 字节！”为此，软件的责任是，保证给硬盘的是512 字节，如果不够，凑也要凑够。因此，513 字节会占用两个扇区，第二个扇区只有一字节是有用的，其他511 字节都是用来填充的。至于某个扇区里，哪些数据是有用的，哪些是填充的，不是硬盘的责任，是软件的责任。就像本章的用户程序一样，通过构造一个头部，自行来跟踪自己的大小。

所以，代码清单8-1 第34 行，判断是否除尽。如果没有除尽，则转移到后面的代码，去读剩余的扇区；如果除尽了，则总扇区数减一。

为什么？为什么除不尽不管，除尽了还要减一？因为刚才已经预读了一个扇区。

注意，用户程序的长度有可能小于512 字节，或者恰好等于512 字节。在这两种情况下，当程序执行到第38 行时，寄存器AX 中的内容必然为零。所以，第38 行是算术比较指令cmp，第39 行是条件转移指令，当寄存器AX 中的内容为零时，就意味着用户程序已经全部读取，不再继续读了，毕竟用户程序只占用一个扇区，而刚才也已经读过了。

用户程序被加载的位置是由DS 和ES 所指向的逻辑段。一个逻辑段最大也才64KB，当用户程序特别大的时候，根本容纳不下。想想看，段内偏移地址从0x0000 开始，一直延伸到最大值0xffff。再大的话，又绕回到0x0000，以至于把最开始加载的内容给覆盖掉了。

其实，要解决这个问题最好的办法是，每次往内存中加载一个扇区前，都重新在前面的数据尾部构造一个新的逻辑段，并把要读取的数据加载到这个新段内。如此一来，因为每个段的大小是512 字节，即，十六进制的0x200，右移4 位（相当于除以16 或者0x10）后是0x20，这就是各个段地址之间的差值。每次构造新段时，只需要在前面段地址的基础上增加0x20 即可得到新段的段地址。

这种做法好有一比，尺子很短，树很高，想只量一次是不可能的，于是只好分几次量，每量一次，将尺子往下挪一挪。

段地址的改变是临时的，毕竟只是为了读取硬盘，所以，代码清单8-1 第42 行，将当前数据段寄存器DS 的内容压栈保存。

第44 行，将用户程序剩余的扇区数传送到寄存器CX，供后面的loop指令使用，因为我们准备采用循环的办法来读完用户程序。

第46～48 行，将当前数据段寄存器DS 的内容在原来的基础上增加0x20，以构造出下一个逻辑段，为从硬盘上读取下一个512 字节的数据做准备。

第50 行，将寄存器BX 清零。BX 被用做数据传输时的段内偏移，而且每次传输都是在一个新的段内进行，故偏移地址在每次传输前都应当是零。

第51 行，每次读硬盘前，将寄存器SI 的内容加一，以指向下一个逻辑扇区。

第52～53 行，调用读硬盘的过程read_hard_disk_0，并开始下一轮循环，直到所有的扇区都读完（寄存器CX 的内容为0）。

8.3.8　用户程序重定位

用户程序在编写的时候是分段的。因此，加载器下一步的工作是计算和确定每个段的段地址。

如图8-16 所示，用户程序定义了6 个段，在编译阶段，编译器为每个段计算了一个汇编地址。第一个段header 位于整个程序的开头，所以其汇编地址为0。从第二个段开始，每个段的汇编地址都是其相对于整个程序开头的偏移量，以字节为单位。因为我们不知道各个段的汇编地址到底是多少，故用字母来表示。这样，第二个段code_1 的汇编地址是v，第三个段code_2 的汇编地址是w，……，最后一个段stack 的汇编地址是z。

现在，用户程序已经全部加载到内存里了，而且是从物理地址phy_base 开始的。如此一来，每个段在内存中的物理地址都是基于phy_base 的，第一个段header 在内存中的起始物理地址是phy_base（phy_base+0），第二个段在内存中的起始物理地址是phy_base+v，……，最后一个段stack 则是phy_base+z。

用于加载用户程序的物理地址phy_base 是16 字节对齐的，而用户程序中，每个段的汇编地址也是16 字节对齐的。因此，每个段在内存中

的起始地址也是16 字节对齐的，将它们分别右移4位，就是它们各自的逻辑段地址。

为此，代码清单8-1 第55 行，从栈中恢复数据段寄存器DS 的内容，使其指向用户程序被加载的起始位置，也就是用户程序头部。

第58～62 行用于重定位用户程序入口点的代码段。请参考图8-15，用户程序头部内，偏移为0x06 处的双字，存放的是入口点代码段的汇编地址。加载器首先将高字和低字分别传送到寄存器DX 和AX，然后调用过程calc_segment_base 来计算该代码段在内存中的段地址。

过程calc_segment_base（计算段基址）是在代码清单8-1 的第134 行定义的。它接受一个32位的汇编地址（位于寄存器DX:AX 中），并在计算完成后向主程序返回一个16 位的逻辑段地址（位于寄存器AX 中）。

因为计算过程中要破坏寄存器DX 的内容，因此，第137 行用于将其压栈保存。

在16 位的处理器上，每次只能进行16 位数的运算。第139 行，先将用户程序在内存中物理起始地址的低16 位加到寄存器AX 中。该指令的地址部分使用了段超越前缀“cs:”，而且也没有加上0x7c00。原因前面已经解释过了，在本程序中，数据段和代码段是分离的，而且当前代码段的定义部分使用了“vstart=0x7c00”子句。

然后，第140 行，再将该起始地址的高16 位加到寄存器DX 中。adc 是带进位加法，它将目的操作数和源操作数相加，然后再加上标志寄存器CF 位的值（0 或者1）。这样，分两步就可以完成32 位数的加法运算。

现在，我们已经在DX:AX 中得到了入口点代码段的起始物理地址，只需要将这个32 位数右移4 位即可得到逻辑段地址。麻烦在于它们分别在两个寄存器中，如何移动？

答案是分别移动，然后拼接。代码清单8-1 第141 行，使用逻辑右移指令shr（SHift logical Right）将寄存器AX 中的内容右移4 位。

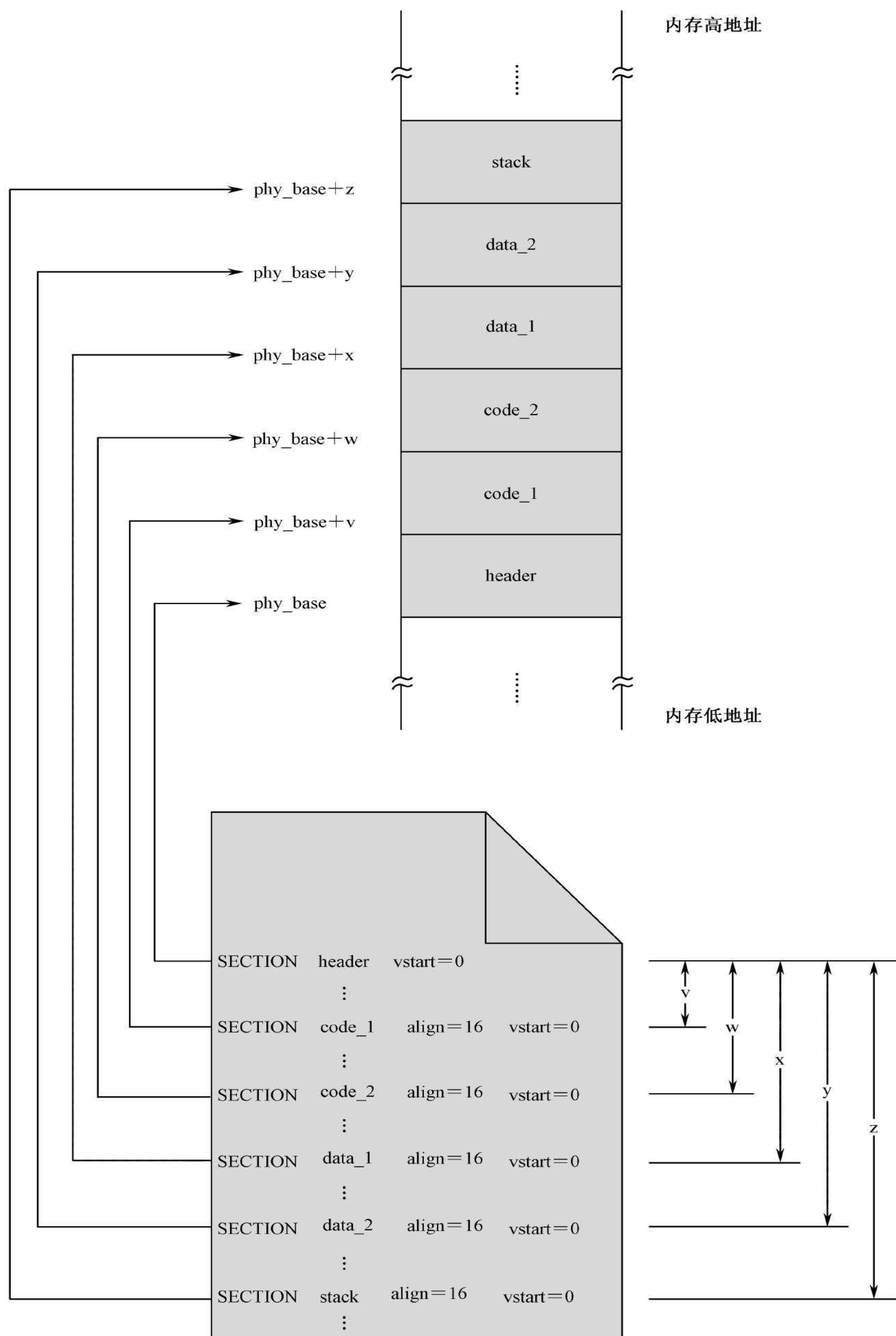


图8-16　段的偏移地址和它在内存中的物理地址

如图8-17 所示，逻辑右移指令执行时，会将操作数连续地向右移动指定的次数，每移动一次，“挤”出来的比特被移到标志寄存器的CF 位，左边空出来的位置用比特“0”填充。

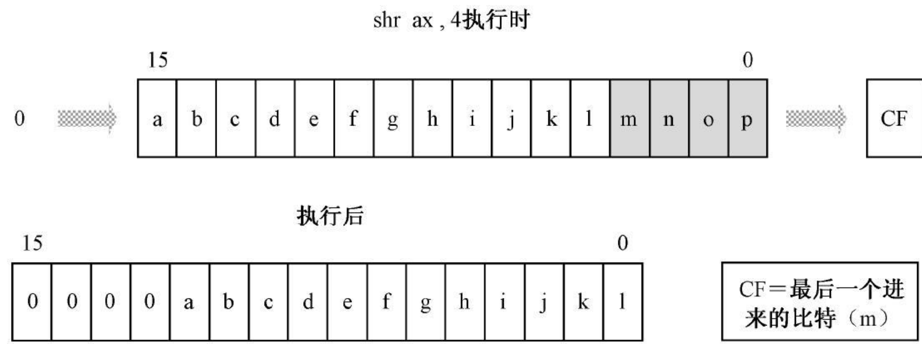


图8-17　逻辑右移示意图

shr 指令的目的操作数可以是8 位或16 位的通用寄存器或者内存单元，源操作数可以是数字1、8 位立即数或者寄存器CL。我们已经介绍过寻址方式，往后，我们要用新的方法来表示指令的格式。就当前指令来说，该指令的格式为：

```
shr r/m8,1        ;目的操作数是 8 位通用寄存器/内存单元，源操作数是 1
shr r/m16,1       ;目的操作数是 16 位通用寄存器/内存单元，源操作数是 1
shr r/m8,imm8     ;目的操作数是 8 位通用寄存器/内存单元，源操作数是 8 位立即数
shr r/m16,imm8    ;目的操作数是 16 位通用寄存器/内存单元，源操作数是 8 位立即数
shr r/m8,cl       ;目的操作数是 8 位通用寄存器/内存单元，源操作数是寄存器 CL
shr r/m16,cl      ;目的操作数是 16 位通用寄存器/内存单元，源操作数是寄存器 CL
```

以上，第一种指令格式的意思是，目的操作数可以是8 位寄存器，或者8 位的内存单元；源操作数是1。对于内存地址的情况，可以使用任何一种我们讲过的内存寻址方式。举三个例子：

```
shr ah,1
shr byte [0x2000],1
shr byte [bx+si+0x02],1
```

第二种指令格式和第一种相似，只是目的操作数的长度不一样。注意，源操作数为1 的逻辑右移指令是特殊设计的优化指令，比如以上的shr ax,1，它的机器码是D1 E8；而类似的指令shr ax,5 则拥有完全不同的机器码C1 E8 05。

第三种指令格式的意思是，目的操作数可以是8 位寄存器，或者8 位的内存单元；源操作数是8 位立即数。下面是两个例子：

```
shr al,0x20                    ;右移 32（0x20）次
shr byte [bx+0x06],0x05        ;右移 5 次
```

第四种指令格式和第二种类似，只是数据宽度不同。

第五种指令格式的目的操作数可以是8 位的寄存器，或者8 位的内存单元；源操作数在寄存器CL 中。如果shr 指令的源操作数是寄存器，则只能使用CL。和一般的指令不同，寄存器CL 只用来提供移动次数，而不用于限定和暗示目的操作数的字长。因此，对于目的操作数是内存地址的情况，必须用关键字byte 或者word 等来加以限定。比如：

```
shr al,cl
shr byte [bx],cl
```

最后一种指令格式适用于目的操作数的长度为字的情况。

注意，和8086 处理器不同，80286 之后的IA-32 处理器在执行本指令时，会先将源操作数的高3 位清零。也就是说，最大的移位次数是31。

shr 的配对指令是逻辑左移指令shl（SHift logical Left），它的指令格式和shr 相同，只不过它是向左移动。

尽管DX:AX 中是32 位的用户程序起始物理内存地址，理论上，它只有20 位是有效的，低16位在寄存器AX 中，高4 位在寄存器DX 的低4 位。寄存器AX 经右移后，高4 位已经空出，只要将DX 的最低4 位挪到这里，就可以得到我们所需要的逻辑段地址。为此，可以使用以下指令：

```
shl dx,12
or ax,dx
```

很显然，代码清单8-1 并不是这么做的，为的是演示另一个不同的指令ror（第142 行），也就是循环右移（ROtate Right）。如图8-18 所示，循环右移指令执行时，每右移一次，移出的比特既送到标志寄存器的CF 位，也送进左边空出的位。

ror 的配对指令是循环左移指令rol（ROtate Left）。ror、rol、shl、shr 的指令格式都是相同的。

因为是循环移位，移位后，寄存器DX 的低12 位是我们不需要的。所以，代码清单8-1 的第143 行，用and 指令将其清零。

第144 行，正式将寄存器AX 和DX 的内容合并，这就是我们要的段地址。

过程的最后，第146～148 行，恢复寄存器DX 的原始内容，并返回到调用程序那里。

现在，回到代码清单8-1 的第62 行，那条指令的功能是将刚刚计算出来的逻辑段地址回写到原处，仅覆盖低16 位，高16 位不用理会。

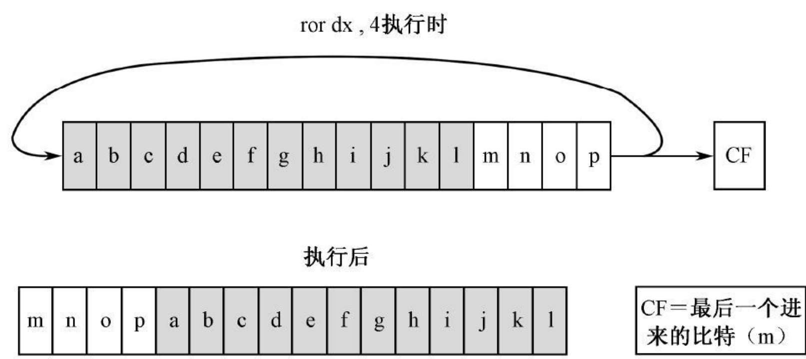


图8-18　循环右移示意图

现在仅仅是处理了入口点代码段的重定位，下面开始正式处理用户程序的所有段，它们位于用户程序头部的段重定位表中。

重定位表的表项数存放在用户程序头部偏移0x0a 处，如图8-5 所示。代码清单8-1 第65 行，用于将它从该内存地址处传送到寄存器CX，供后面的循环指令使用。

段重定位表的首地址存放在用户程序头部偏移0x0c 处，因此，第66行，将0x0c 传送到基址寄存器BX 中。以后，每次只要将BX 的内容加上4，就指向下一个重定位表项。

第68～74 行是循环体，每次循环开始后，BX 总是指向需要重定位的段的汇编地址，而且都是双字，需要分别传送到寄存器DX 和AX。然后调用过程calc_segment_base 计算相应的逻辑段地址，并覆盖到原来的位置（低字），最后将基址寄存器的内容加上4 ，以指向下一个表项。当寄存器CX 的内容为0 时，循环结束，所有的段都处理完毕。

8.3.9　将控制权交给用户程序

现在，用户程序已经在内存中准备就绪，剩下的工作就是把处理器的控制权交给它。交接工作很简单，代码清单8-1 第76 行，加载器通过一个16 位的间接绝对远转移指令，跳转到用户程序入口点。

如图8-15 所示，入口点是两个连续的字，低字是偏移地址，位于用户程序头部内偏移为0x04的地方；高字是段地址，位于用户程序头部内偏移为0x06 的地方。而且，因为加载器的辛勤工作，该段地址是已经重定位过的。

处理器执行指令

```
jmp far [0x04]
```

时，会访问段寄存器DS 所指向的数据段，从偏移地址为0x04 的地方取出两个字，并分别传送到代码段寄存器CS 和指令指针寄存器IP，以替代它们原先的内容。于是，处理器就像被洗脑了一样，自行转移到指定的位置处开始执行。

处理器已经跑到用户程序内部去执行了，所以接下来的工作是跟踪用户程序的工作流程。不过，在此之前，还是先总结一下无条件转移指令jmp 的用法。

8.3.10　8086 处理器的无条件转移指令

1. 相对短转移

相对短转移的操作码为0xEB，操作数是相对于目标位置的偏移量，仅1 字节，是个有符号数。由于这个原因，该指令属于段内转移指令，而且只允许转移到距离当前指令-128～127 字节的地方。相对短转移指令必须使用关键字“short”。例如：

```
jmp short infinite
```

在源程序编译阶段，编译器会检查标号infinite 所代表的值，如果数值超过了一字节所能允许的数值范围，则无法通过编译。否则，编译器

用目标位置的汇编地址减去当前指令的汇编地址，再减去当前指令的长度（2），保留1 字节的结果，作为机器指令的操作数。

相对短转移指令的汇编语言操作数只能是标号和数值。下面是直接使用数值的情况：

```
jmp short 0x2000
```

但数值和标号是等价的。在编译阶段，都被用来计算一个8 位的偏移量。

在指令执行时，处理器把指令中的操作数加上2，再加到指令指针寄存器IP 上，这会导致指令的执行流程转向目标地址处。

2．16 位相对近转移

和相对短转移不同，16 位相对近转移指令的转移范围稍大一些。它的机器指令操作码为0xE9，而且，该指令的长度为3 字节，操作码0xE9 后面还有一个16 位（2 字节）的操作数。

因为是近转移，故其属于段内转移。“相对”的意思同样是指它的操作数是一个相对量，是相对于目标位置处的偏移量。在源程序编译阶段，编译器用目标位置的汇编地址减去当前指令的汇编地址，再减去当前指令的长度（3），保留16 位的结果，作为机器指令的操作数。由于这是一个16 位的有符号数，故可以转移到距离当前指令-32768～32767 字节的地方。

16 位相对近转移指令应当使用关键字“near”，比如

```
jmp near infinite
jmp near 0x3000
```

在早先的NASM 版本中，关键字near 是可以省略的。若没有指定short 或者near，那么，编译器自动默认是“near”的。但是最近的版本改变了这一规则。如果没有指定关键字short 或者near，那么，如果目标位置距离当前指令-128～127 字节，则自动采用short；否则，采用near。

3．16 位间接绝对近转移

这种转移方式也是近转移，即只在段内转移。但是，转移到的目标偏移地址不是在指令中直接给出的，而是用一个16 位的通用寄存器或者内存地址来间接给出的。比如：

```
jmp near bx
```

```
jmp near cx
```

指令中的关键字“near”可以省略，间接绝对近转移原本就是near的。以上两条指令执行时，处理器将用寄存器BX 或者CX 的内容来取代指令指针寄存器IP 的当前内容。

以上是目标偏移地址位于通用寄存器的情况。当然，该偏移地址也可位于内存中，而且这是最常见的情况。假如在某程序的数据段中声明了标号jump_dest 并初始化了一个字：

```
jump_dest dw 0xc000
```

而且假定我们已经知道它是转移目标的起始偏移地址，那么，在该程序的代码段内，就可以使用以下的16 位间接绝对近转移指令：

```
jmp [jump_dest]                ;省略关键字“near”，本小节内下同
```

当这条指令执行时，处理器访问由段寄存器DS 指向的数据段，从指令中指定的偏移地址处取得一个字（在这里是0xc000），并用该字取代指令指针寄存器IP 的当前内容。

当然，既然是间接地寻找目标位置的偏移地址，其他寻址方式也是可以的。比如：

```
jmp [bx]
jmp [bx+si]
```

注意，jmp bx 和jmp [bx]是完全不同的，不要犯迷糊。前者，要转移的绝对偏移地址位于寄存器BX 中；后者，偏移地址位于由BX 所指向的内存字单元中。

4．16 位直接绝对远转移

很早以前，我们曾经见过这样的指令：

```
jmp 0x0000:0x7c00
```

在这里，0x0000 和0x7c00 分别是段地址和偏移地址，符合“段地址：偏移地址”的表达习惯。在编译之后，其机器指令为

```
EA 00 7C 00 00
```

0xEA 是操作码，后面是操作数。注意，字的存放是按照低端字节序的。而且，在编译之后，偏移地址在前，段地址在后。执行这条指令后，处理器用指令中给出的段地址代替段寄存器CS 的原有内容，用给出的偏移地址代替IP 寄存器的原有内容，从而跳转到另一个不同的代码段中，即执行一个段间转移。

像这种直接在指令中给出段地址和偏移地址的转移指令，就是直接绝对远转移指令。“16 位”仅仅用来限定偏移地址部分，指偏移地址是16 位的。

5．16 位间接绝对远转移（jmp far）

远转移的目标地址可以通过访问内存来间接得到，这叫间接远转移，但是要使用关键字“far”。假如在某程序的数据段内声明了标号jump_far，并在其后初始化了两个字：

```
jump_far dw 0x33c0,0xf000
```

这不是两个普通的数值，它们分别是某个程序片断的偏移地址和段地址。为了转移到该程序片断上执行，可以在使用下面的转移指令：

```
jmp far [jump_far]
```

关键字“far”的作用是告诉编译器，该指令应当编译成一个远转移。处理器执行这条指令后，访问段寄存器DS 所指向的数据段，从指令中给出的偏移地址处取出两个字，分别用来替代段寄存器CS 和指令指针寄存器IP 的内容。

其实，最好的例子还是本章代码清单8-1 的第76 行：

```
jmp far [0x04]
```

16 位间接绝对远转移指令的操作数可以是任何一种内存寻址方式。除了上面的例子外，下面再给出几个：

```
jmp far [bx]
jmp far [bx+si]
```

最后，“16 位”的意思是，要转移到的目标位置的偏移地址是16 位的。

检测点8.3

1．以下指令执行后，寄存器AX 中的内容是多少？

```
mov ax,0x55aa
ror ax,8
shr ax,2
```

2．按题目的要求写出相应的指令：

a．无条件转移到当前段内标号label_proc 处；

b．无条件转移到当前段内的另一个位置，偏移地址在寄存器BX 中；

c．无条件转移到当前段内的另一个位置，偏移地址保存在当前附加段内由寄存器BX 所指向的内存单元中；

d．无条件转移，段地址为0xf000，偏移地址为0x0002；

e．无条件转移，段地址和偏移地址存放在当前数据段内偏移地址为0x80 的地方，低字是目标处的偏移地址，高字为目标处段地址；

f．无条件转移，段地址和偏移地址存放在当前附加段内，低字为目标的偏移地址，高字为目标的段地址，这两个字在当前附加段内的偏移地址可以用BX+DI+0x08 得到。

8.4 用户程序的工作流程

8.4.1 初始化段寄存器和栈切换

现在轮到用户程序在处理器上执行了。

用户程序的入口点在代码清单8-2 的第135 行。因为加载器已经完成了重定位工作，所以用户程序的头等大事是初始化处理器的各个段寄存器DS、ES、SS，以便访问专属于自己的数据。段寄存器CS 就不用初始化了，那是加载器负责做的事。要不然用户程序怎么可能执行呢。

在刚刚进入用户程序时，段寄存器DS 和ES 依然指向段header，而栈段寄存器SS 依然指向加载器的栈空间。代码清单8-2 的第137、138 行，用于从头部取得用户程序自己的栈段的段地址，并传送到段寄存器SS 中。

第139 行，将标号stack_end 所代表的数值传送到栈指针寄存器SP。该标号是在第205 行声明的，在它的前面，是伪指令resb，用来保留256 字节的栈空间。

伪指令resb（REServe Byte）的意思是从当前位置开始，保留指定数量的字节，但不初始化它们的值。在源程序编译时，编译器会保留一段内存区域，用来存放编译后的内容。当它看到这条伪指令时，它仅仅是跳过指定数量的字节，而不管里面的原始内容是什么。内存是反复使用的，谁也无法知道以前的使用者在这里留下了什么。也就是说，跳过的这段空间，每个字节的值是不确定的。

因此，

```
        resb 256
```

将在编译后的内容中保留256 字节。resb 不是唯一用来声明未初始化数据的指令。以下是另外一些：

```
        resw 100                ;声明 100 个未初始化的字
        resd 50                 ;声明 50 个未初始化的双字
```

栈段stack 的定义中有“vstart=0”子句，保留的256 字节，其汇编地址分别是0～255。所以，标号stack_end 处的汇编地址实际上是256。也就是说，代码清单8-2 的第139 行和以下指令等价：

```
mov sp,256
```

栈切换完毕之后，第141、142 行，从用户程序头部取得数据段data_1 的段地址，传送到段寄存器DS 中。从此，DS 不再指向段header，不能再用它访问用户程序头部了。

据此也可以看出，各个段寄存器的初始化顺序很重要。如果先初始化数据段和附加段，那么，段header 中的数据将无法访问。

8.4.2 调用字符串显示例程

紧接着，用户程序要在屏幕上显示东西了。

要显示的内容位于段data_1 中，该段当前正由段寄存器DS 指向。代码清单8-2 第175 行，声明了标号msg0 并初始化了一大堆字符。当然，因为字符太多，行太长，而我们还想能大致“看”到显示效果，所以分成了多行来初始化。

为太长的行使用续行符“\”当然是一个好主意，不过我们现在的做法是将太长的行分成几段，分别用伪指令db 来初始化。在编译之后，它们仍然是紧挨在一起的，可以用唯一的标号msg0 来引用。

在屏幕上显示字符，所做的仅仅是填充显存，只要所填充的内容不超过一屏所能显示的字符数，其他的事不需要你操心。当字符在一行上显示不下时，显示系统会自动移到下一行接着显示，这也和你无关。

不过，有时候我们希望有自行换行的能力，而不管那一行是否已经到头（屏幕最右边）。这么做的目的通常是用来格式化文本段落。

再来回顾一下ASCII 码。在128 个ASCII 代码中，大部分是可显示和打印的字符，还有一部分用于控制显示和打印那些字符的设备。比如0x0d 是回车，0x0a 是换行。

回车和换行的概念最早起源于老式打字机。那种打字机上有滚筒，用于使纸张上下卷动，每敲击一个按键，字车往右移动一格，位于下一个可打印的位置。在这种古老而不失先进性的设备上，将字车推到最左

边，也就是一行的开始，叫做回车（Carriage Return）；而拧一下滚筒，将纸上卷一行，叫做换行（Line Feed）。如果既回车，又换行，那么，字车将位于下一行的行首。这个过程通常叫做回车换行（CRLF）。

在刚刚有了电子计算机的时候，因为它又大又贵，只能通过远程终端来分享它的计算能力。这时候，用的是电传打字机，不需要人工操作即可显示和打印字符。当然，根据需要随时回车换行还是需要的。怎么办？那就是用ASCII 码中的控制字符来命令电传打字机来做这件事。不知怎么回事，回车分配的ASCII 码是0x0d，换行分配的则是0x0a。奇怪吗？没什么好奇怪的。

在个人计算机时代，为了在屏幕上显示字符，ASCII 码也被引入显示系统。不过，当我们向显存里写入0x0d 和0x0a 时，并不起任何作用，也没有任何效果，没有任何硬件对解释它们的意义负责。不过无所谓，对回车换行代码的解释可以由我们自己负责，现在所要做的，就是在字符串中，需要回车换行的地方按照老传统插入这两个代码。

正是由于以上的原因，在代码清单8-2 的第175～191 行，凡是需要回车换行的地方，都使用了0x0d 和0x0a。而且，在第191 行，也就是所有要显示内容最后，是数值0，用来标志字符串的结束，这样的字符串称为是0 终止的字符串，在高级语言里经常使用。

段data_1 的定义中包括“vstart=0”子句，故标号msg0 的汇编地址是从该段的起始处（0）开始计算的。代码清单8-2 的第144、145 行，将该字符串的偏移地址传送到基址寄存器BX，并调用过程put_string。

8.4.3 过程的嵌套

过程put_string 是在当前代码段定义的，位于代码清单8-2 的第28 行，用于显示给定的字符串。它接受两个参数DS 和BX，分别是字符串所在的段地址和偏移地址。另外，它要求字符串的最后一个数值是0，作为终止的标记。

过程put_string 的工作很简单，它循环从DS:BX 中取得单个字符，判断它是否为0。不为0 则调用另一个过程put_char，为0 则返回主程序。

为此，代码清单8-2 第30 行，从当前数据段中取得一个字符，段地址在DS 中，偏移地址由BX 提供。

第31 行，通过or 指令来促成标志的产生，它的功能类似于

```
cmp cl,0
```

在这里，or 指令的两个操作数相同，都是寄存器CL，一个数和它自己做“或”运算，结果还是它自己，但计算结果会影响标志寄存器中的某些位。如果ZF 置位，说明取到了串结束标志0，转移到第38 行返回主程序；否则，将取到的字符作为参数调用另一个过程put_char。

当过程put_char 返回后，第34 行，将寄存器BX 的内容加一以指向下一个要显示的字符。

第35 行，无条件转移到当前过程的开始处，重复取字符过程。

允许在一个过程中调用另一个过程，这称为过程嵌套。因为每次调用过程时，处理器都把返回地址压在栈中，返回时从栈中取得返回地址，所以，只要栈是安全的，嵌套的过程都能层层返回。

过程嵌套的层数在原则上是没有限制的，唯一的限制是栈的大小。不要忘了，实模式下，栈的空间最大是64KB，每执行一次过程调用需要2 字节或4 字节，这还没有包括在每个过程内部消耗的栈空间。

8.4.4 屏幕光标控制

过程put_char 用于显示一个字符。但它与常规方法的不同之处在于，它能判断回车和换行，还能在超过屏幕上最后一行的时候上滚内容，就是我们经常说的卷屏或者滚屏。除此之外，它还使用了光标跟随技术。

光标（Cursor）是在屏幕上有规律地闪动的一条小横线，通常用于指示下一个要显示的字符位置，这对很多年龄比较大的人来说很熟悉（前提是他们以前也用过计算机）。在那个时代，还没有基于图形显示技术的Windows，所有的软件都在文本模式下工作，而基于硬件的光标只在文本模式下才会出现。

计算机技术发展得很快，很多硬件都已经或者即将淘汰，但显卡是个例外。即使是现在，多年前形成的VGA 显示标准在每块显卡中都完好地保留下来了，包括对光标的支持。原因很简单，在显卡中集成一块支持128 个ASCII 代码的字符发生器非常方便，在程序中显示一个字符也只

要给出它的ASCII 码。显示图形的代价太大，在计算机加电启动的时候，以及其他一些根本没必要、也没条件使用图形模式的场合，这是最好的选择。

光标在屏幕上的位置保存在显卡内部的两个光标寄存器中，每个寄存器是8 位的，合起来形成一个16 位的数值。比如，0 表示光标在屏幕上第0 行第0 列，80 表示它在第1 行第0 列，因为标准VGA 文本模式是25 行，每行80 个字符。这样算来，当光标在屏幕右下角时，该值为25×80－1=1999。

光标寄存器是可读可写的。你可以从中读出光标的位置，也可以通过它设置光标的位置。能够通过写入一个数值来设定光标的位置，这不是恩赐，而是责任，因为显卡从来不自动移动光标位置，这个任务是你的。现在你总算明白为什么它是可写的了吧？

8.4.5　取当前光标位置

显卡的操作非常复杂，内部的寄存器也不是一般地多。为了不过多占用主机的I/O 空间，很多寄存器只能通过索引寄存器间接访问。

索引寄存器的端口号是0x3d4，可以向它写入一个值，用来指定内部的某个寄存器。比如，两个8 位的光标寄存器，其索引值分别是14（0x0e）和15（0x0f），分别用于提供光标位置的高8 位和低8 位。

指定了寄存器之后，要对它进行读写，这可以通过数据端口0x3d5来进行。

好，现在言归正传。过程put_char 看起来并不太复杂，但实际上判断和分支较多。为了便于读者理解这段代码，也为了方便讲解，图8-19给出了它的工作流程图。

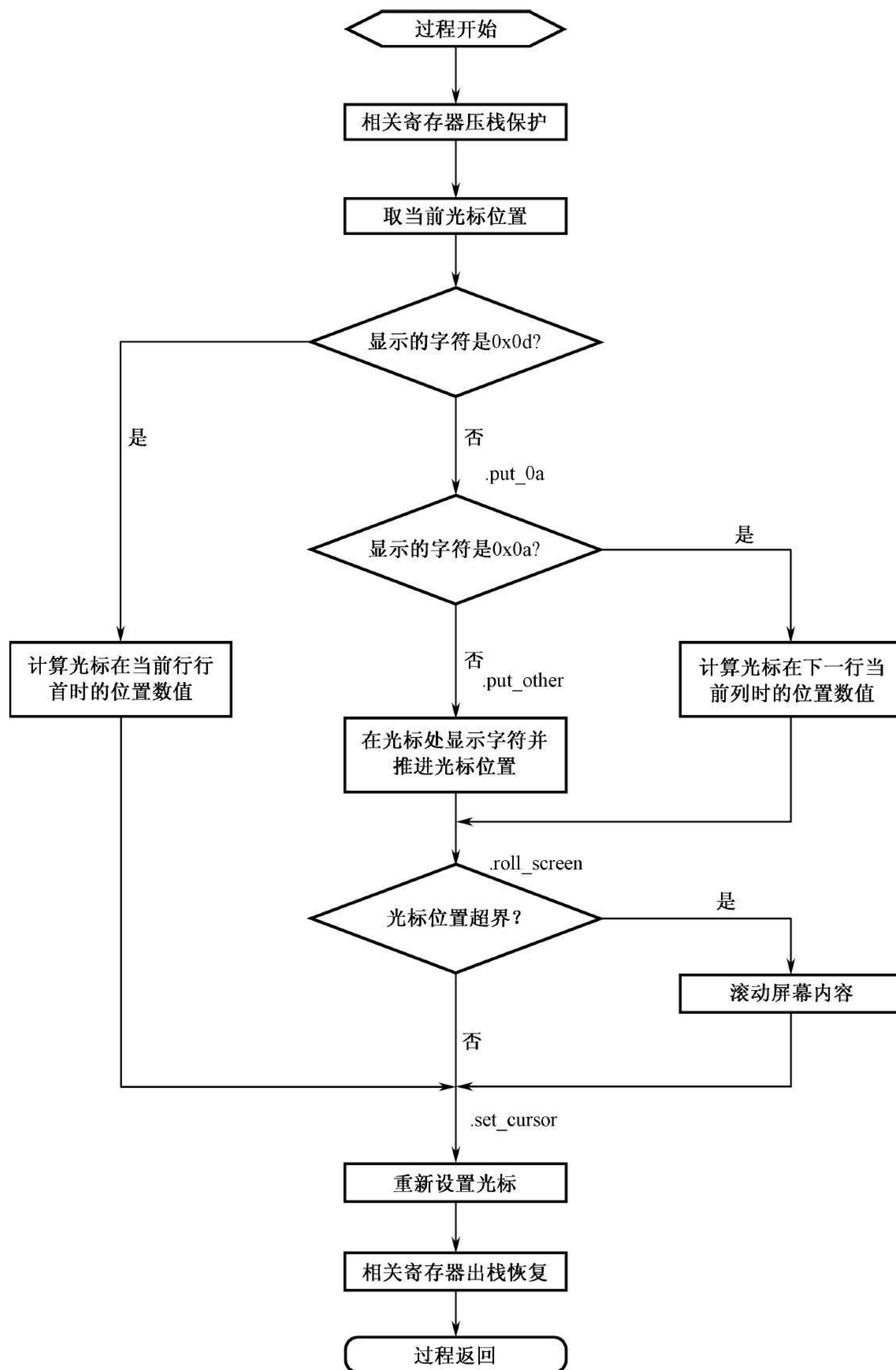


图8-19　过程put_char 的流程图

庞大复杂的建筑必须得有图纸才能施工，而绘制图纸的过程中你经常发现自己有更好的设计思路，更能知道如何盖这栋房子。编写复杂的程序前先画一画流程图，是程序员的基本素养，这有助于问题的解决。

代码清单8-2 第43～48 行，在过程put_char 的开始部分先将用到的部分寄存器压栈保存，其中包括两个段寄存器DS 和ES。

第51～53 行，通过索引端口告诉显卡，现在要操作0x0e 号寄存器。

第54～56 行，通过数据端口从0x0e 号端口读出1 字节的数据，并传送到寄存器AH 中，这是屏幕光标位置的高8 位。

同样地，第58～62 行，从0x0f 号寄存器读出光标位置的低8 位。现在，寄存器AX 中是完整的光标位置数据。第63 行，将这个数值传送到寄存器BX 中保存，因为马上就要用到寄存器AX。

8.4.6 处理回车和换行字符

过程put_char 仅接受一个寄存器参数CL，用于提供要显示的ASCII 码。常规字符和回车、换行符将不同对待，为此，需要首先别出它们。

代码清单8-2 第65、66 行，先判断是不是回车符0x0d。如果是的话，继续往下执行，如果不是，则转移到标号.put_0a 处执行。

先来看看如果是0x0d 的情况。

如果是回车符0x0d，那么，应将光标移动到当前行的行首。每行有80 个字符，那么，用当前光标位置除以80，余数不要，就可以得到当前行的行号。接着，再乘以80，就是当前行行首的光标数值。

很好，代码清单8-2 第67～69 行，用寄存器AX 中的光标位置除以寄存器BL 中的80，在AL 中得到的是当前行的行号。

接着，第70、71 行，将寄存器AL 中的内容乘以寄存器BL 中的80，会在寄存器AX 中得到当前行行首的光标值。该值依然传送到寄存器BX 中保存。

和div 指令相反，mul 是乘法指令，格式如下：

```
mul  r/m8              ;AX=AL×r/m8
mul  r/m16             ;DX:AX=AX×r/m16
```

以上，“r”表示通用寄存器，“m”表示内存单元。就是说，mul 指令可以用8 位的通用寄存器或者内存单元中的数和寄存器AL 中的内容相乘，

结果是16 位，在AX 寄存器中；也可以用16位的通用寄存器或者内存单元中的数和寄存器AX 中的内容相乘，结果是32 位，高16 位和低16位分别在DX 和AX 中。

举几个例子：

```
mul bx
mul dx
mul byte [bx]          ;8 位内存单元
mul byte [bx+di]       ;8 位内存单元
mul word [0x2000]      ;16 位内存单元
```

mul 指令执行后，要是结果的高一半为全0，则OF 和CF 清零，否则置1。对SF、ZF、AF 和PF 标志的影响未定义。

第72 行，转移到标号.set_cursor 处设置光标在屏幕上的位置。

如果要显示的字符不是0x0d，那么，它有可能是0x0a，或者是正常的可打印字符。这里的“打印”，可以理解为在屏幕上打印。

为此，第75～77 行，先判断是不是0x0a，如果不是，那就转移到标号.put_other 处，去正常显示可打印字符。如果是，那么，换行的意图是向下挪一行，只需要将寄存器BX 的内容增加80，即可得到新的光标位置数据。但是，不像回车，如果光标原先就在屏幕最后一行，那么，换行之后，会怎样呢？所以，第78 行，立即转移到标号.roll_screen 处执行。在那里，将根据情况决定是否需要滚屏。

8.4.7　显示可打印字符

下面开始正常显示可打印字符。

第81、82 行，将附加段寄存器ES 设置为指向显存。注意，在过程开始处，已经将ES 的内容压栈保存了，这里可以随意使用该寄存器。

标准模式下，屏幕上可以同时显示2000 个字符。光标占用一个字符的位置，但整个屏幕只有一个，只能出现在2000 个字符位置中的一个上。典型地，程序员要用光标位置来记载和跟踪下一个字符应当显示在什么位置。光标用来指示字符位置，而一个字符在显存中对应两个字节。如此一来，可以将光标位置乘以2，来得到该位置（字符）在显存中的偏移地址。

第83 行，将寄存器BX 的内容逻辑左移1 次，这相当于将其乘以2。毕竟只是乘以2，而且BX 中的数值不大，这样做，比使用乘法指令mul来得方便。

第84 行，用BX 的内容作为偏移地址，来访问段寄存器ES 所指向的显存，来写入要显示的字符。你可能觉得奇怪，为什么后面没有写显示属性字节。原因很简单，在写入其他内容之前，显存里全是黑底白字的空白字符，所以不需要重写黑底白字的属性。过程put_char 是以黑底白字来显示字符的。

第87、88 行，将寄存器BX 的内容除以2，恢复它的光标位置身份。接着，将其增加1（在数值上，将光标推进到下一个位置，毕竟还没开始设置光标呢）。指令shr 是已经讲过的逻辑右移指令，相当于除以2。

不管是换行，还是正常显示字符后推进光标，都会使寄存器BX 的内容超过1999。下面，就来判断这个情况，并决定是否滚动屏幕内容。

8.4.8 滚动屏幕内容

第91、92 行，比较寄存器BX 中的内容是否小于2000。如果是的话，很好，很正常，直接转移到标号.set_cursor 处设置光标；否则继续往下执行以滚动屏幕内容。

滚动屏幕内容，实质上就是将屏幕上第2～25 行的内容整体往上提一行，最后用黑底白字的空白字符填充第25 行，使这一行什么也不显示。

为了加快速度，提高效率，程序里采用的是将数据从一个内存区域（块）搬运到另一个内存区域（块）的做法，核心指令是movsw。

第94～101 行，设定源区域从显存内偏移地址为0xa0（屏幕第2 行第1 列的位置）的地方开始，该区域的段地址在段寄存器DS 中，偏移地址在变址寄存器SI 中；目标区域从显存内偏移地址为0x00（屏幕第1 行第1 列的位置）的地方开始，该区域的段地址在段寄存器ES 中，偏移地址在变址寄存器DI 中。同时，设置方向标志，并在寄存器CX 中设置要传送的字数1920（24 行乘以80 个字符/行，再乘以每个字符占用的字节数2，再除以2 字节/字）。最后，执行rep movsw以完成传送工作。

屏幕最下面一行（第25 行）还有原来的内容，必须予以清除。第25 行第1 列在显存中的偏移地址是3840。为此，第102～107 行，使用黑底白字的空白字符循环写入这一行。

最后，第109 行，滚屏之后，光标应当位于最后一行的第1 列，其数值为1920，这一行的指令将这个新的数值传送到寄存器BX 中。

8.4.9 重置光标

不管是回车、换行，还是显示可打印的字符，上面的各处都给出了光标位置的新数值。下面的工作就是按给出的数值在屏幕上设置光标。

第112～123 行，还是依照老规矩，通过索引端口指定光标寄存器0x0e 和0x0f，并分别将寄存器BX 中的高8 位和低8 位通过数据段口0x3d5 写入它们。

最后，第125～130 行，从栈中依次弹出并恢复各个寄存器的原始内容。

第132 行，指令ret 从栈中恢复指令指针寄存器IP 的内容，返回到调用者put_string 过程。当字符串msg0 中所有的字符都显示完毕后，过程put_string 返回到用户主程序，从第147 行接着往下执行。

8.4.10 切换到另一个代码段中执行

在一个程序中，对段的数量没有限制。可以有多个代码段和多个数据段，甚至可以有多个栈段。在用户程序工作时，可以从一个代码段转到另一个代码段中执行，也可以根据需要，访问不同的数据段。

我们知道，ret 和retf 指令分别用于近返回和远返回。人类最大的问题就是思维有定势，有时候不够开阔。尽管说是“返回”，但最重要的还是弄清它的原理和本质，才能灵活运用。

返回指令的动作是从栈中弹出内容到指令指针寄存器IP，如果是远返回的话，还要接着弹出内容到代码段寄存器CS。假如在此之前，栈顶的内容并非是用于返回的偏移地址和段地址，那么处理器当时就会傻了。

还是回到正题上来。假如要想切换到另一个代码段中执行，可以使用远调用指令（call far）或者远转移指令（jmp far），这是最正常不过的途径了。

问题在于，为了实现段间控制转移，必须事先开辟两个连续的内存单元，存放另一个代码段的入口点偏移地址和段地址，代价似乎有点高，这么做好像不太值得。

为了省事，可以使用指令retf 来模拟段间返回，以实现段间转移。代码清单8-2 第147 行，先在栈中压入代码段code_2 的段地址；接着，第148、149 行，压入偏移地址，该偏移地址就是标号begin 在编译阶段的汇编地址。8086 处理器不能在栈中压入立即数，所以只能通过寄存器AX来间接做这件事，现在的处理器都支持压入立即数：

```
push word 0x55ff
```

当然，这是后话。

第151 行，当处理器执行指令retf 时，这个被蒙在鼓里的家伙从栈中将偏移地址和段地址分别弹出到代码段寄存器CS 和指令指针寄存器IP，于是控制立即转移到段code_2 中，从标号begin处开始执行。

这段代码很好地证明了，尽管call 和call far 指令分别依赖于ret 和retf 指令，但后者却并不依赖于前者。它们经常在一起，但并不是夫妻。

8.4.11　访问另一个数据段

你可以在代码段code_2 中做任何事。但是，我们这里什么也没干，仅仅是用相同的方法，再次返回到段code_1 中。具体的做法，可以参考代码清单8-2 第166～170 行。

回到第154、155 行，由于自从进入用户程序之后，段寄存器ES 一直是指向头部段header的，所以，这两条指令用于将第二个数据段data_2 的段地址传送到段寄存器DS，这等于是换了一个数据段。

第二个数据段data_2 是在第194 行定义的，而且包含了“vstart=0”子句。在该段内，仅仅声明了标号msg1 并初始化了一个字符串。当然，它也是0 结尾的。

接着回到前面的第157 行，将刚才那个字符串的起始偏移地址传送到寄存器BX。第158 行，调用过程put_string 从屏幕的光标处开始显示该字符串。

8.5 编译和运行程序并观察结果

通常，用户程序执行完毕后，应当重新将控制返回到加载器，加载器可以重新加载和运行其他程序，所有的操作系统都是这么做的。

遗憾的是，我们的加载器不提供这样的功能，而用户程序也没有将控制返回到加载器，而是直接进入无限循环：

```
        jmp $
```

当然，这不是什么了不得的事情，将控制返回到加载器，其实现也不复杂。如果你有兴趣，可以试一试。但是，唯一麻烦的地方是栈，将控制返回的同时，也必须切换到加载器自己的栈，一定要小心！

对本章源代码的讲解到此结束。你可以在配书工具中找到源代码c08_mbr.asm 和c08.asm，或者自己手工编辑这两个文件。

首先编译源程序c08_mbr.asm，将编译后得到的c08_mbr.bin 文件写入虚拟硬盘主引导扇区（逻辑0 扇区）。然后，编译源程序c08.asm，并将生成的c08.bin 文件写入虚拟硬盘的逻辑100 扇区。

注意，在编译c08.asm 时，编译器将会产生警告信息：

```
c08.asm:203: warning: uninitialized space declared in stack section: zeroing
```

这句话的意思是，c08.asm 源程序的第203 行声明了未初始化的空间。

还记得吗？在那里，我们用resb 伪指令保留了256 字节的栈空间，这段空间是未初始化的。

源程序的编译过程也是排错过程，你该感到高兴，而不是害怕，只有合乎规范的程序才能最终获得通过。编译器通常会有两种提示，一种是错误，另一种是警告。

错误（Error）表明程序中有编译器不认识的指令、不正确的语法和无法解释的内容，在这种情况下，编译器简单地告诉你是哪一行有错误，以及什么性质的错误，并停止编译。

警告（Warning）通常表示程序中有一些不规范的指令用法。在这种情况下，编译器继续完成编译工作，生成编译结果。通常情况下，编译后的结果也能正常运行。

现在，启动虚拟机，正常情况下，运行结果应当如图8-20 所示。

```
LEARN-ASM [正在运行] - Oracle VM VirtualBox
控制(M)  View  设备(D)  帮助(H)
 This is NASM - the famous Netwide Assembler. Back at SourceForge and in intens
ive development! Get the current versions from http://www.nasm.us/.

 Example code for calculate 1+2+...+1000:

    xor dx,dx
    xor ax,ax
    xor cx,cx
 @@:
    inc cx
    add ax,cx
    adc dx,0
    inc cx
    cmp cx,1000
    jle @@
    ... ...(Some other codes)

 The above contents is written by LeeChung. 2011-05-06_
Right Ctrl
```

图8-20　本章程序运行结果

本章习题

1．修改本章源程序8-2，在不使用retf 指令的情况下，从段code_1 转移到段code_2 执行。

2．思考一下，如果去掉代码清单8-1 的第38、39 行，会发生什么情况？

第9章　中断和动态时钟显示

在享受计算机给我们带来的便利和乐趣的同时，我仍然会时不时地说它的坏话。人们都说处理器是整个计算机的大脑，可是，处理器是一个非常精确，速度又快的傻子。

在计算机上执行的程序通常需要一些输入，输入可能来自于键盘、鼠标、硬盘、话筒、数码相机等，同时，处理后还需要输出，要送到输出设备，如显示器、硬盘、打印机、网络设备等。

一个程序只做自己的事，当它等待输入，或者等待输出时，它面对的是比处理器慢得多的外部设备。典型的情况下，硬盘的工作速度比处理器至少慢几千万甚至几亿倍，像打印机这类设备就更不用说了。在等待的时候，处理器唯一所能做的，就是不停地观察外部设备的状态变化。

计算机革命的早期，硬件资源极其昂贵和稀少。据说20 世纪60 年代，一台计算机的价格抵得上 300 辆野马跑车，月租金超过一万美金。这么昂贵的东西，不好好利用它就是一种罪过。

为了分享计算能力，处理器应当能够为多用户多任务提供硬件一级的支持。在单处理器的系统中，允许同时有多个程序在内存中等待处理器的执行。当一个程序正在等待输入输出时，允许另一个程序从处理器那里得到执行权。

如何把多个程序调入内存，是操作系统的事情，这个可以先放一放。现在的问题是，当一个程序执行时，它是不会知道还有别的程序正眼巴巴地等着执行。在这种情况下，中断（Interrupt）这种工作机制就应运而生了。

中断就是打断处理器当前的执行流程，去执行另外一些和当前工作不相干的指令，执行完之后，还可以返回到原来的程序流程继续执行。这就好比是你正在用手机听歌，突然来电话了。处理器（当然，手机也是有处理器的）必须中断歌曲的播放，来处理这件更为重要的事件。

自从中断这种工作机制产生之后，它就一直是各种处理器必须具备的机制。中断是怎么发生的，处理器又是怎么处理中断的，在这个过程

中，我们又能做些什么，这都是本章将要告诉你的。总起来说，本章的任务是：

1． 了解中断的原理和分类，用两个具体的实例来学习如何在中断机制下工作，包括如何使用BIOS 中断工作。

2． 学会在Bochs 中观察中断向量表和中断标志位IF 的变化。

3． 学习一些新的x86 处理器指令，包括into、int3、int n、iret、cli、sti、hlt、not 和test 等。

9.1 外部硬件中断

顾名思义，外部硬件中断，就是从处理器外面来的中断信号。当外部设备发生错误，或者有数据要传送（比如，从网络中接收到一个针对当前主机的数据包），或者处理器交给它的事情处理完了（比如，打印已经完成），它们都会拍一下处理器的肩膀，告诉它应当先把手头上的事情放一放，来临时处理一下。

如图9-1 所示，外部硬件中断是通过两个信号线引入处理器内部的。从很早的时候起，也就是8086 处理器的时代，这两根线的名字就叫NMI和INTR。

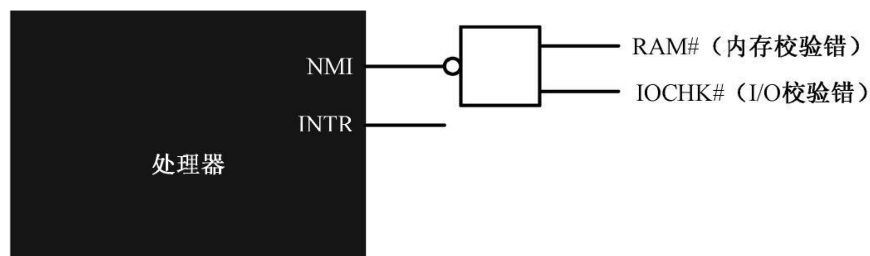


图9-1 Intel 处理器上的不可屏蔽中断示意图

9.1.1 非屏蔽中断

在某些具有怀疑精神的人眼里，用两根信号线来接受外部设备中断可能是多余的，也许只需要一根就可以了。这似乎有此道理，但是，来自外部设备的中断很多，也不是每一个中断都是必须处理的。有些中断，在任何时候都必须及时处理，因为事关整个系统的安全性。比如，在使用不间断电源的系统中，当电池电量很低的时候，不间断电源系统会发出一个中断，通知处理器快掉电了。再比如，内存访问电路发现了一个校验错误，这意味着，从内存读取的数据是错误的，处理器再努力工作也是没有意义的。在所有这些情况下，处理器必须针对这些中断采取必要的措施，隐瞒真相必然会对用户造成不可挽回的损失。除此之外，更多的中断是可以被忽略或者延迟处理的，如果某个程序希望不被打扰的话。

在这种情况下，处理器的设计者希望通过两个引脚来明确区分不同性质的中断，这是很自然的事。首先，所有的严重事件都必须无条件地加以处理，这种类型的中断是不会被阻断和屏蔽的，称为非屏蔽中断（Non Maskable Interrupt，NMI）。

中断信号的来源，或者说，产生中断的设备，称为中断源。如图9-1所示，在传统的兼容模式下，NMI 的中断源通过一个与非门连接到处理器。处理器的NMI 引脚是高电平有效的，而中断信号是低电平有效的。当不存在中断的时候，与非门的所有输入都为高，因此处理器的NMI 引脚为低电平，这意味着没有中断发生。

当有任何一个非屏蔽的中断产生时，与非门的输出为高。Intel 处理器规定，NMI 中断信号由0 跳变到1 后，至少要维持4 个以上的时钟周期才算是有效的，才能被识别。

注意，不要把这幅图当成是不变的真理，这是一个简化的示意图，不是真正的设备连接图。

当一个中断发生时，处理器将会通过中断引脚NMI 和INTR 得到通知。除此之外，它还应当知道发生了什么事，以便采取适当的处理措施。每种类型的中断都被统一编号，这称为中断类型号、中断向量或者中断号。但是，由于不可屏蔽中断的特殊性——几乎所有触发NMI 的事件对处理器来说都是致命的，甚至是不可纠正的。在这种情况下，努力去搞清楚发生了什么，通常没有太大的意义，这样的事最好留到事后，让专业维修人员来做。

也正是这个原因，在实模式下，NMI 被赋予了统一的中断号2，不再进行细分。一旦发生2号中断，处理器和软件系统通常会放弃继续正常工作的“念头”，也不会试图纠正已经发生的问题和错误，很可能只是由软件系统给出一个提示信息。

9.1.2　可屏蔽中断

和NMI 不同，更多的时候，发往处理器的中断信号通常不会意味着灾难。当然，有时候也会非常紧急，比如，在一个由计算机控制的车床上，当零件快速通过铣具时，处理器应当立即处理中断，并向铣具发送信号，告诉它应当如何切削。

这类中断有两个特点，第一是数量很多，毕竟有很多外部设备；第二是它们可以被屏蔽，这样处理器就像是没听见、没看见一样，不会对它们进行处理。所以，这类硬件中断称为可屏蔽中断。尽管不处理中断就会把零件铣坏，但是否允许处理器看见该中断，是你自己的事，这是处理器赋予你的权利。

可屏蔽中断是通过INTR 引脚进入处理器内部的，像NMI 一样，不可能为每一个中断源都提供一个引脚。而且，处理器每次只能处理一个中断。在这种情况下，需要一个代理，来接受外部设备发出的中断信号。还有，多个设备同时发出中断请求的几率也是很高的，所以该代理的任务还包括对它们进行仲裁，以决定让它们中的哪一个优先向处理器提出服务请求。

如图9-2 所示，在个人计算机中，用得最多的中断代理就是8259 芯片，它就是通常所说的中断控制器，从8086 处理器开始，它就一直提供着这种服务。即使是现在，在绝大多数单处理器的计算机中，也依然有它的存在。

Intel 处理器允许256 个中断，中断号的范围是0～255，8259 负责提供其中的15 个，但中断号并不固定。之所以不固定，是因为当初设计的时候，允许软件根据自己的需要灵活设置中断号，以防止发生冲突。该中断控制器芯片有自己的端口号，可以像访问其他外部设备一样用in 和out 指令来改变它的状态，包括各引脚的中断号。正是因为这样，它又叫可编程中断控制器（Programmable Interrupt Controller，PIC）。

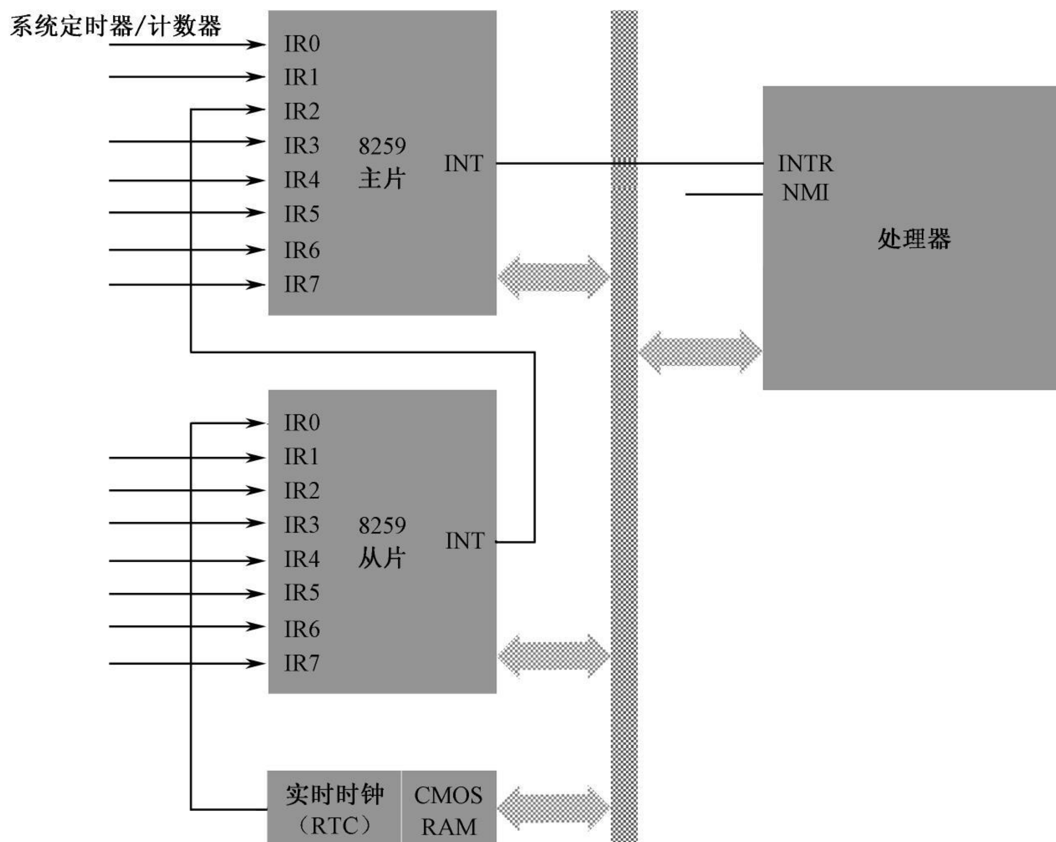


图9-2　单处理器系统的中断机制

不知道是怎么想的，反正每片8259 只有8 个中断输入引脚，而在个人计算机上使用它，需要两块。如图9-2 所示，第一块8259 芯片的代理输出INT 直接送到处理器的INTR 引脚，这是主片（Master）；第二块8259 芯片的INT 输出送到第一块的引脚2 上，是从片（Slave），两块芯片之间形成级联（Cascade）关系。

如此一来，两块8259 芯片可以向处理器提供15 个中断信号。当时，接在8259 上的15 个设备都是相当重要的，如PS/2 键盘和鼠标、串行口、并行口、软磁盘驱动器、IDE 硬盘等。现在，这些设备很多都已淘汰或者正在淘汰中，根据需要，这些中断引脚可以被其他设备使用。

如图9-2 所示，8259 的主片引脚0（IR0）接的是系统定时器/计数器芯片；从片的引脚0 （IR0）接的是实时时钟芯片RTC，该芯片是本章的主角，很快就会讲到。总之，这两块芯片的固定连接即使是在硬件更新换代非常频繁的今天，也依然没有改变。

在8259 芯片内部，有中断屏蔽寄存器（Interrupt Mask Register，IMR），这是个8 位寄存器，对应着该芯片的8 个中断输入引脚，对应的

位是0 还是1，决定了从该引脚来的中断信号是否能够通过8259 送往处理器（0 表示允许，1 表示阻断，这可能出乎你的意料）。当外部设备通过某个引脚送来一个中断请求信号时，如果它没有被IMR 阻断，那么，它可以被送往处理器。注意，8259 芯片是可编程的，主片的端口号是0x20 和0x21，从片的端口号是0xa0 和0xa1，可以通过这些端口访问8259 芯片，设置它的工作方式，包括IMR 的内容。

中断能否被处理，除了要看8259 芯片的脸色外，最终的决定权在处理器手中。回到前面第6章，参阅图6-2，你会发现，在处理器内部，标志寄存器有一个标志位IF，这就是中断标志（Interrupt Flag）。当IF 为0时，所有从处理器INTR 引脚来的中断信号都被忽略掉；当其为1 时，处理器可以接受和响应中断。

IF 标志位可以通过两条指令cli 和sti 来改变。这两条指令都没有操作数，cli（CLear Interrupt flag）用于清除IF 标志位，sti（SeT Interrupt flag）用于置位IF 标志。

检测点9.1

写一个小的主引导程序，在程序中使用sti 和cli 指令，并用Bochs 观察IF 位的变化。

在计算机内部，中断发生得非常频繁，当一个中断正在处理时，其他中断也会陆续到来，甚至会有多个中断同时发生的情况，这都无法预料。不用担心，8259 芯片会记住它们，并按一定的策略决定先为谁服务。总体上来说，中断的优先级和引脚是相关的，主片的IR0 引脚优先级最高，IR7 引脚最低，从片也是如此。当然，还要考虑到从片是级联在主片的IR2 引脚上。

最后，当一个中断事件正在处理时，如果来了一个优先级更高的中断事件时，允许暂时中止当前的中断处理，先为优先级较高的中断事件服务，这称为中断嵌套。

9.1.3 实模式下的中断向量表

所谓中断处理，归根结底就是处理器要执行一段与该中断有关的程序（指令）。处理器可以识别256 个中断，那么理论上就需要256 段程序。这些程序的位置并不重要，重要的是，在实模式下，处理器要求将它们的入口点集中存放到内存中从物理地址0x00000 开始，到0x003ff 结

束，共1KB 的空间内，这就是所谓的中断向量表（Interrupt Vector Table，IVT）。

如图9-3 所示，每个中断在中断向量表中占2 个字，分别是中断处理程序的偏移地址和段地址。中断0 的入口点位于物理地址0x00000 处，也就是逻辑地址0x0000:0x0000；中断1 的入口点位于物理地址0x00004 处，即逻辑地址0x0000:0x0004；其他中断以此类推，总之是按顺序的。

当中断发生时，如果从外部硬件到处理器之间的道路都是畅通的，那么，处理器在执行完当前的指令后，会立即着手为硬件服务。它首先会响应中断，告诉8259 芯片准备着手处理该中断。接着，它还会要求8259 芯片把中断号送过来。

在8259 芯片那里，每个引脚都赋予了一个中断号。而且，这些中断号是可以改变的，可以对8259 编程来灵活设置，但不能单独进行，只能以芯片为单位进行。比如，可以指定主片的中断号从0x08 开始，那么它每个引脚IR0～IR7 所对应的中断号分别是0x08～0x0e。

中断信号来自哪个引脚，8259 芯片是最清楚的，所以它会把对应的中断号告诉处理器，处理器拿着这个中断号，要顺序做以下几件事。

① 保护断点的现场。首先要将标志寄存器FLAGS 压栈，然后清除它的IF 位和TF 位。TF 是陷阱标志，这个以后再讲。接着，再将当前的代码段寄存器CS 和指令指针寄存器IP 压栈。

② 执行中断处理程序。由于处理器已经拿到了中断号，它将该号码乘以4（毕竟每个中断在中断向量表中占4 字节），就得到了该中断入口点在中断向量表中的偏移地址。接着，从表中依次取出中断程序的偏移地址和段地址，并分别传送到IP 和CS，自然地，处理器就开始执行中断处理程序了。

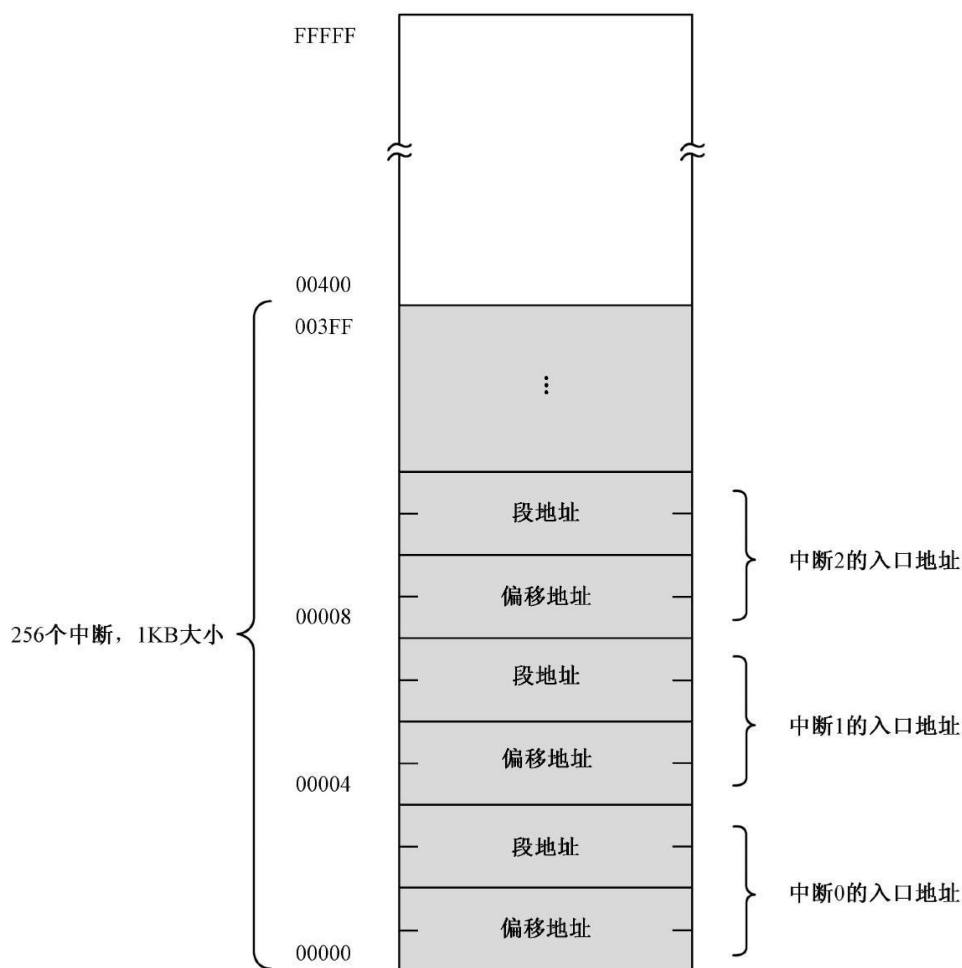


图9-3　实模式下的中断向量表

注意，由于IF 标志被清除，在中断处理过程中，处理器将不再响应硬件中断。如果希望更高优先级的中断嵌套，可以在编写中断处理程序时，适时用sti 指令开放中断。

③ 返回到断点接着执行。所有中断处理程序的最后一条指令必须是中断返回指令iret。这将导致处理器依次从栈中弹出（恢复）IP、CS 和FLAGS 的原始内容，于是转到主程序接着执行。

iret 同样没有操作数，执行这条指令时，处理器依次从栈中弹出数值到IP、CS 和标志寄存器。

顺便提醒一句，由于中断处理过程返回时，已经恢复了FLAGS 的原始内容，所以IF 标志位也自动恢复。也就是说，可以接受新的中断。

和可屏蔽中断不同，NMI 发生时，处理器不会从外部获得中断号，它自动生成中断号码2，其他处理过程和可屏蔽中断相同。

中断随时可能发生，中断向量表的建立和初始化工作是由BIOS 在计算机启动时负责完成的。BIOS 为每个中断号填写入口地址，因为它不知道多数中断处理程序的位置，所以，一律将它们指向一个相同的入口地址，在那里，只有一条指令：iret。也就是说，当这些中断发生时，只做一件事，那就是立即返回。当计算机启动后，操作系统和用户程序再根据自己的需要，来修改某些中断的入口地址，使它指向自己的代码。马上你就会看到，我们在本章也是这样做的。

9.1.4　实时时钟、CMOS RAM 和BCD 编码

也许你曾经觉得奇怪，为什么计算机能够准确地显示日期和时间？原因很简单，如图9-2 所示，在外围设备控制器芯片ICH 内部，集成了实时时钟电路（Real Time Clock，RTC）和两小块由互补金属氧化物（CMOS）材料组成的静态存储器（CMOS RAM）。实时时钟电路负责计时，而日期和时间的数值则存储在这块存储器中。

实时时钟是全天候跳动的，即使是在你关闭了计算机的电源之后，原因在于它由主板上的一个小电池提供能量。它为整台计算机提供一个基准时间，为所有需要时间的软件和硬件服务。不像8259 芯片，有关RTC CMOS 的资料相当少见，很不容易完整地找到，而8259 的内容则铺天盖地，到处都是。所以，本章只是简要地介绍8259，而尽量多说一些和RTC 有关的知识。

早期的计算机没有ICH 芯片，各个接口单元都是分立的，单独地焊在主板上，并彼此连接。早期的RTC 芯片是摩托罗拉（Motorola）MS146818B，现在直接集成在ICH 内，并且在信号上与其兼容。除了日期和时间的保存功能外，RTC 芯片也可以提供闹钟和周期性的中断功能。

日期和时间信息是保存在CMOS RAM 中的，通常有128 字节，而日期和时间信息只占了一小部分容量，其他空间则用于保存整机的配置信息，比如各种硬件的类型和工作参数、开机密码和辅助存储设备的启动顺序等。这些参数的修改通常在BIOS SETUP 开机程序中进行。要进入该程序，一般需要在开机时按DEL、ESC、F1、F2 或者F10 键。具体按哪个键，视计算机的厂家和品牌而定。

RTC 芯片由一个振荡频率为32.768kHz 的石英晶体振荡器（晶振）驱动，经分频后，用于对CMOS RAM 进行每秒一次的时间刷新。

如表9-1 所示，常规的日期和时间信息占据了CMOS RAM 开始部分的10 字节，有年、月、日和时、分、秒，报警的时、分、秒用于产生到时间报警中断，如果它们的内容为0xC0～0xFF，则表示不使用报警功能。

表9-1　CMOS RAM 中的时间信息

偏移地址	内　　容	偏移地址	内　　容
0x00	秒	0x07	日
0x01	闹钟秒	0x08	月
0x02	分	0x09	年
0x03	闹钟分	0x0A	寄存器 A
0x04	时	0x0B	寄存器 B
0x05	闹钟时	0x0C	寄存器 C
0x06	星期	0x0D	寄存器 D

CMOS RAM 的访问，需要通过两个端口来进行。0x70 或者0x74 是索引端口，用来指定CMOS RAM 内的单元；0x71 或者0x75 是数据端口，用来读写相应单元里的内容。举个例子，以下代码用于读取今天是星期几：

```
mov al,0x06
out 0x70,al
in al,0x71
```

不得不说的是，从很早的时候开始，端口0x70 的最高位（bit 7）是控制NMI 中断的开关。当它为0 时，允许NMI 中断到达处理器，为1 时，则阻断所有的NMI 信号，其他7 个比特，即0～6位，则实际上用于指定CMOS RAM 单元的索引号，这种规定直到现在也没有改变。

如图9-4 所示，尽管端口0x70 的位7 不是中断信号，但它能控制与非门的输出，决定真正的NMI 中断信号是否能到达处理器。

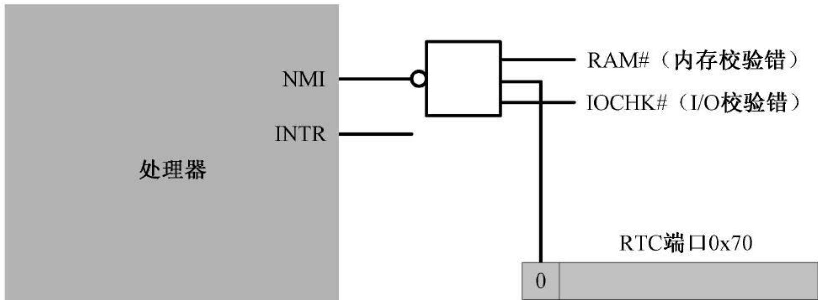


图9-4　端口0x70 的位7 用于禁止或允许NMI（仅为示意图）

通常来说，在往端口0x70 写入索引时，应当先读取0x70 原先的内容，然后将它用于随后的写索引操作中。但是，该端口是只写的，不能用于读出。在早期的系统中，计算机的制造成本很高，为了最大化地利用硬件资源，导致出现很多稀奇古怪的做法，这就是一个活生生的例子。

为了解决这个问题，同时也为了兼容以前的老式硬件，ICH 芯片允许通过切换访问模式来临时取得那些只写寄存器的内容，但这涉及更高层次的知识，已经超出了当前的话题范畴。现在，我们只想把问题搞得简单些，这么说吧，NMI 中断应当始终是允许的，在访问RTC 时，我们直接关闭NMI，访问结束后，再打开NMI，而不管它以前到底是什么样子。

在早期，CMOS RAM 只有64 字节，而最新的ICH 芯片内则可能集成了256 字节，新增的128字节称为扩展的CMOS RAM。当然，在此之前，要先确保ICH 内确实存在扩展的CMOS RAM。

CMOS RAM 中保存的日期和时间，通常是以二进制编码的十进制数（Binary Coded Decimal，BCD），这是默认状态，如果需要，也可以设置成按正常的二进制数来表示。要想说明什么是BCD 编码，最好的办法是举个例子。比如十进制数25，其二进制形式是00011001。但是，如果采用BCD 编码的话，则一个字节的高4 位和低4 位分别独立地表示一个0 到9 之间的数字。因此，十进制数25 对应的BCD 编码是00100101。由此可以看出，因为十进制数里只有0～9，故用BCD 编码的数，高4 位和低4 位都不允许大于1001，否则就是无效的。

单元0x0A～0x0D 不是普通的存储单元，而被定义成4 个寄存器的索引号，也是通过0x70 和0x71 这两个端口访问的。这4 个寄存器用于设置实时时钟电路的参数和工作状态。

寄存器A 和B 用于对RTC 的功能进行整体性的设置，它们都是8 位的寄存器，可读可写，其各位的用途如表9-2 和表9-3 所示。

表9-2　寄存器A 各位功能说明

比 特 位	功 能
7	正处于更新过程中（Update In Progress，UIP）。该位可以作为一个状态进行监视。CMOS RAM 中的时间和日期信息会由 RTC 周期性地更新，在此期间，用户程序不应当访问它们。对当前寄存器的写入不会改变此位的状态 0：更新周期至少在 488 微秒内不会启动。换句话说，此时访问 CMOS RAM 中的时间、日历和闹钟信息是安全的 1：正处于更新周期，或者马上就要启动 如果寄存器 B 的 SET 位不是 1，而且在分频电路已正确配置的情况下，更新周期每秒发生一次。在此期间，会增加保存的日期和时间、检查数据是否因超出范围而溢出（比如，31 号之后是下月 1 号，而不是 32 号），还要检查是否到了闹钟时间，最后，更新之后的数据还要写回原来的位置 更新周期至少会在 UIP 置 1 后的 488μs 内开始，而且整个周期的完成时间不会多于 1984 μs，在此期间，和日期时间有关的存储单元（0x00～0x09）会暂时脱离外部总线。为避免更新和数据遭到破坏，可以有两次安全地从外部访问这些单元的机会：当检测到更新结束中断发生时，可以有差不多 999ms 的时间用于读写有效的日期和时间数据；如果检测到寄存器 A 的 UIP 位为低（0），那么这意味着在更新周期开始前，至少还有 488μs 的时间

续表

比 特 位	功 能
6～4	分频电路选择（Division Chain Select）。这 3 位控制晶体振荡器的分频电路。系统将其初始化到 010，为 RTC 选择一个 32.768kHz 的时钟频率
3～0	速率选择（Rate Select，RS）。选择分频电路的分节点。如果寄存器 B 的 PIE 位被设置的话，此处的选择将产生一个周期性的中断信号，否则将设置寄存器 C 的 PF 标志位 0000：从不触发中断 0001：3.90625 ms 0010：7.8125 ms 0011：122.070 μs 0100：244.141 μs 0101：488.281 μs 0110：976.5625 μs 0111：1.953125 ms 1000：3.90625 ms 1001：7.8125 ms 1010：5.625 ms 1011：1.25 ms 1100 = 62.5 ms 1101：125 ms 1110：250 ms 1111：500 ms

表9-3　寄存器B 各位功能说明

比 特 位	功 能
7	更新周期禁止（Update Cycle Inhibit，SET）。允许或者禁止更新周期 0：更新周期每秒都会正常发生 1：中止当前的更新周期，并且此后不再产生更新周期。此位置 1 时，BIOS 可以安全地初始化日历和时间
6	周期性中断允许（Periodic Interrupt Enable，PIE） 0：不允许 1：当达到寄存器 A 中 RS 所设定的时间基准时，允许产生中断
5	闹钟中断允许（Alarm Interrupt Enable，AIE） 0：不允许 1：允许更新周期在到达闹点并将 AF 置位的同时，发出一个中断
4	更新结束中断允许（Update-Ended Interrupt Enable，UIE） 0：不允许 1：允许在每个更新周期结束时产生中断
3	方波允许（Square Wave Enable，SQWE） 该位空着不用，只是为了和早期的 Motorola 146818B 实时时钟芯片保持一致
2	数据模式（Data Mode，DM） 该位用于指定二进制或者 BCD 的数据表示形式 0：BCD 1：Binary
1	小时格式（Hour Format，HOURFORM） 0：12 小时制。在这种模式下，第 7 位为 0 表示上午（AM），为 1 表示下午（PM） 1：24 小时制
0	老软件的夏令时支持（Daylight Savings Legacy Software Support，DSLSWS） 该功能已不再支持，该位仅用于维持对老软件的支持，并且是无用的

寄存器C 和D 是标志寄存器，这些标志反映了RTC 的工作状态，寄存器C 是只读的，寄存器D 则可读可写，它们也都是8 位寄存器，其各位的含义如表9-4 和表9-5 所示。特别是寄存器C，因为RTC 可以产生中断，当中断产生时，可以通过该寄存器来识别中断的原因，比如，是周期性的中断，还是闹钟中断。

表9-4　寄存器C 各位功能说明

比 特 位	功 能
7	中断请求标志（Interrupt Request Flag，IRQF） IRQF＝（PF×PIE）＋（AF×AIE）＋（UF×UFE） 以上，加号表示逻辑或，乘号表示逻辑与。该位被设置时，表示肯定要发生中断。对寄存器 C 的读操作将导致此位清零
6	周期性中断标志（Periodic Interrupt Flag，PF）。 若寄存器 A 的 RS 位为 0000，则此位是 0，否则是 1。对寄存器 C 的读操作将导致此位清零 注：程序可以根据此位来判断 RTC 的中断原因
5	闹钟标志（Alarm Flag，AF）。 当所有闹点同当前时间相符时，此位是 1。对寄存器 C 的读操作将导致此位清零 注：程序可以根据此位来判断 RTC 的中断原因
4	更新结束标志（Update-Ended Flag，UF） 紧接着每秒一次的更新周期之后，RTC 电路立即将此位置 1。对寄存器 C 的读操作将导致此位清零 注：程序可以根据此位来判断 RTC 的中断原因
3～0	保留，总是报告 0

表9-5 寄存器D 各位功能说明

比 特 位	功 能
7	有效 RAM 和时间位（Valid RAM and Time Bit，VRT） 在写周期，此位应当始终写 0。不过，在读周期，此位回到 1。在 RTC 加电正常时，此位被硬件强制为 1
6	保留。总是返回 0。并且在写周期总是置 0
5～0	日期闹钟（Date Alarm），这些位保存着闹钟的月份数值

讲了这么多和8259 以及RTC 有关的内容，现在，我们想让RTC 芯片定期发出一个中断，当这个中断发生的时候，还能执行我们自己编写的代码，来访问CMOS RAM，在屏幕上显示一个动态走动的时钟。

9.1.5 代码清单9-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。
本章代码清单：9-1（被加载的用户程序），源程序文件：c09_1.asm

9.1.6 初始化8259、RTC 和中断向量表

本章提供的代码清单中，没有加载器程序。这是因为可以利用上一章提供的加载器来加载用户程序，只要符合规则，加载器是通用的。

用户程序的入口点在代码清单9-1 的第119 行，从这一行开始，到第124 行，用于初始化各个段寄存器的内容。下面开始在中断向量表中安

装实时时钟中断的入口点。既然本章的主题是中断，那么就很有必要强调一件事。当处理器执行任何一条改变栈段寄存器SS 的指令时，它会在下一条指令执行完期间禁止中断。

栈无疑是很重要的，不能被破坏。要想改变代码段和数据段，只需要改变段寄存器就可以了。但栈段不同，因为它除了有段寄存器，还有栈指针。因此，绝大多数时候，对栈的改变是分两步进行的：先改变段寄存器SS 的内容，接着又修改栈指针寄存器SP 的内容。

想象一下，如果刚刚修改了段寄存器SS，在还没来得及修改SP 的情况下，就发生了中断，会出现什么后果，而且要知道，中断是需要依靠栈来工作的。

因此，处理器在设计的时候就规定，当遇到修改段寄存器SS 的指令时，在这条指令和下一条指令执行完毕期间，禁止中断，以此来保护栈。换句话说，你应该在修改段寄存器SS 的指令之后，紧跟着一条修改栈指针SP 的指令。

就代码清单9-1 来说，在第121、122 行执行期间，处理器禁止中断。再比如以下指令：

```
push cs
pop ss
mov sp,0
```

在后面两行指令执行期间，处理器禁止中断。

RTC 芯片的中断信号，通向中断控制器8259 从片的第1 个中断引脚IR0。在计算机启动期间，BIOS 会初始化中断控制器，将主片的中断号设为从0x08 开始，将从片的中断号设为从0x70开始。所以，计算机启动后，RTC 芯片的中断号默认是0x70。尽管我们可以通过对8259 编程来改变它，但是没有必要。

检测点9.2

在Bochs 中使用“xp”命令显示实模式下的中断向量表，并找出0x70 号中断处理过程的段地址和偏移地址。

在安装中断向量之前，应该先显示些什么。第126～130 行，显示两行提示信息，表明正在安装中断向量。这两个字符串位于第286 行的数

据段中。对于过程put_string 没有什么好说的，它的代码和上一章相同，工作过程更没有区别。

为了修改某中断在中断向量表中的登记项，需要先找到它。第132～135 行，将中断号0x70乘以4，就是它在中断向量表内的偏移。

第137 行，修改中断向量表时，需要先用cli 指令清中断。当表项信息只修改了一部分时，如果发生0x70 号中断，则会产生不可预料的问题。

第139～141 行，将段寄存器ES 压栈暂时保存，并使它指向中断向量表（所在的段）。

接着，第142～145 行，访问中断向量表内0x70 号中断的表项，分别写入新中断处理过程的偏移地址和段地址。新的中断处理过程是从标号new_int_0x70 处开始的，而且位于当前代码段内。所以，该中断处理过程的偏移地址就是标号new_int_0x70 的汇编地址（注意，段code 的定义中带有vstart=0 子句），段地址就是当前段寄存器CS 的内容。表项修改完毕，从栈中恢复段寄存器ES 的原始内容。

接下来，我们要设置RTC 的工作状态，使它能够产生中断信号给8259 中断控制器。

RTC 到8259 的中断线只有一根，而RTC 可以产生多种中断。比如闹钟中断、更新结束中断和周期性中断（参见表9-3 和表9-4）。RTC 的计时（更新周期）是独立的，产生中断信号只是它的一个赠品。所以，如果希望它能产生中断信号，需要额外设置。

以上所说的三种中断，我们只要设置一种就可以了。其实，最简单的就是设置更新周期结束中断。每当RTC 更新了CMOS RAM 中的日期和时间后，将发出此中断。更新周期每秒进行一次，因此该中断也每秒发生一次。

为了设置该中断，代码清单9-1 第147 行，将RTC 寄存器B 的索引0x0b 传送到寄存器AL。在访问RTC 期间，最好是阻断NMI，因此，第148、149 行，先用or 指令将AL 的最高位置1，再写端口0x70。

第150、151 行，用于通过数据端口0x71 写寄存器B。写的内容是0x12，其二进制形式为00010010，对照表9-3，其意义不难理解：允许更新周期照常发生，禁止周期性中断，禁止闹钟功能，允许更新周期结束中断，使用24 小时制，日期和时间采用BCD 编码。

每次当中断实际发生时，可以在程序（中断处理过程）中读寄存器C的内容来检查中断的原因。比如，每当更新周期结束中断发生时，RTC就将它的第4 位置1。该寄存器还有一个特点，就是每次读取它后，所有内容自动清零。而且，如果不读取它的话（换句话说，相应的位没有清零），同样的中断将不再产生。

为此，第153～155 行，读一下寄存器C 的内容，使之开始产生中断信号。注意，在向索引端口0x70 写入的同时，也打开了NMI。毕竟，这是最后一次在主程序中访问RTC。

当然，如果采用周期性中断而不是更新周期结束中断，则稍微麻烦一些，因为要设置分频电路的分节点。以下代码片断用于产生2 次/秒的周期性中断：

```
mov al,0x0a
or al,0x80
out 0x70,al
in al,0x71
or al,0x0f                    ;设置 RTC 寄存器 A，使其每秒发生 2 次中断
out 0x71,al
```

除此之外，还要设置寄存器B 的PIE 位，以允许周期性中断。

RTC 芯片设置完毕后，再来打通它到8259 的最后一道屏障。正常情况下，8259 是不会允许RTC 中断的，所以，需要修改它内部的中断屏蔽寄存器IMR。IMR 是一个8 位寄存器，位0 对应着中断输入引脚IR0，位7对应着引脚IR7，相应的位是0 时，允许中断，为1 时，关掉中断。

8259 芯片是我见过的芯片中，访问起来最麻烦，也是我最讨厌的一个。好在有关它的资料非常好找，这里就简单地进行讲解。代码清单9-1第157～159 行，通过端口0xa1 读取8259 从片的IMR 寄存器，用and 指令清除第0 位，其他各位保持原状，然后再写回去。于是，RTC 的中断可以被8259 处理了。

第161 行，sti 指令将标志寄存器的IF 位置1，开放设备中断。从这个时候开始，中断随时都会发生，也随时会被处理。

9.1.7　使处理器进入低功耗状态

RTC 更新周期结束中断的处理过程可以看成另一个程序，是独立的处理过程，是额外的执行流程，它随时都会发生，但和主程序互不相干。关于它的执行过程，马上就要讲到，现在继续来看主程序。

在为中断过程做了初始化工作之后，主程序还是要继续执行的。代码清单9-1 第163～167行，用于显示中断处理程序已安装成功的消息。

接着，第169～171 行，使段寄存器DS 指向显示缓冲区，并在屏幕上的第12 行33 列显示一个字符“@”，该位置差不多是整个屏幕的中心。表达式12*160 + 33*2 是在指令编译阶段计算的，是该字符在显存中的位置。每个字符在显存中占2 字节的位置，每行80 个字符。

在此之后，主程序就无事可做了。第174 行，hlt 指令使处理器停止执行指令，并处于停机状态，这将降低处理器的功耗。处于停机状态的处理器可以被外部中断唤醒并恢复执行，而且会继续执行hlt 后面的指令。

所以，第174～176 行用于形成一个循环，先是停机，接着某个外部中断使处理器恢复执行。一旦处理器的执行点来到hlt 指令之后，则立即使它继续处于停机状态。

第175 行，使用not 指令将字符@的显示属性反转。not 是按位取反指令，其格式为

```
not r/m8
not r/m16
```

not 指令执行时，会将操作数的每一位反转，原来的0 变成1，原来的1 变成0。比如：

```
mov al,0x1f
not al          ;执行后，AL 的内容为 0xe0
```

从显示效果上看，循环将显示属性反转将取得一个动画效果，可以很清楚地看到处理器每次从停机状态被唤醒的过程。not 指令不影响任何标志位。

相对于jmp $指令，使用hlt 指令会大大降低处理器的占用率。Windows 7 操作系统有一个叫做CPU 仪表盘的小工具，当使用jmp $指令时，你会看到处理器占用率是100%；而在一个循环中使用hlt 指令

时，该占用率马上降到10%左右，这还是在虚拟机环境下，毕竟宿主操作系统还要占用处理器时间。

9.1.8 实时时钟中断的处理过程

主程序就是这样了，停机、执行，接着停机。与此同时，中断也在不停地发生着，处理器还要抽出空来执行中断处理过程，下面就来看看RTC 的更新周期结束中断处理，该中断处理过程从代码清单9-1 的第27行开始。

第28～32 行，先保护好现场，将后面用到的寄存器压栈保存。这一点特别重要，中断处理过程必须无痕地执行，你不知道中断会在什么时候发生，也不知道中断发生时，哪一个程序正在执行，所以，必须保证中断返回时，能还原中断前的状态。

第34～40 行，用于读RTC 寄存器A，根据UIP 位的状态来决定是等待更新周期结束，还是继续往下执行。UIP 位为0 表示现在访问CMOS RAM 中的日期和时间是安全的。注意第36 行，用于把寄存器AL 的最高位置1，从而阻断NMI。当然，这是不必要的，当NMI 发生时，整个计算机都应当停止工作，也不在乎中断处理过程能否正常执行。

第38 行从数据端口读取寄存器A 的内容；第39 行，test 指令用于测试寄存器AL 的第7 位是否为1。

“test”的意思是“测试”。顾名思义，可以用这条指令来测试某个寄存器，或者内存单元里的内容是否带有某个特征。

test 指令在功能上和and 指令是一样的，都是将两个操作数按位进行逻辑“与”，并根据结果设置相应的标志位。但是，test 指令执行后，运算结果被丢弃（不改变或破坏两个操作数的内容）。

test 指令需要两个操作数，其指令格式为

```
test r/m8,imm8
test r/m16,imm16
test r/m8,r8
test r/m16,r16
```

和and 指令一样，test 指令执行后，OF＝CF＝0；对ZF、SF 和PF 的影响视测试结果而定；对AF 的影响未定义。对于test 指令的应用，这

里有一个例子，比如，我们想测试Al 寄存器的第3 位是“0”还是“1”，可以这样编写代码：

```
test al,0x08
```

0x08 的二进制形式为00001000，它的第3 位是“1”，表明我们关注的是这一位。不管寄存器AL 中的内容是什么，只要它的第3 位是“0”，这条指令执行后，结果一定是00000000，标志位ZF＝1；

相反，如果寄存器AL 的第3 位是“1”，那么结果一定是00001000，ZF＝0。于是，根据ZF 标志位的情况，就可以判定寄存器AL 中的第3 位是“0”还是“1”。

第40 行，如果UIP 位是0，那么测试的结果是ZF＝1，继续往下执行第42 行；否则，说明UIP 位是1，需要返回到第34 行继续等待RTC 更新周期结束。

正常情况下，访问CMOS RAM 中的日期和时间，必须等待RTC 更新周期结束，所以上面的判断过程是必需的，而这些代码也适用于正常的访问过程。但是，当前中断处理过程是针对更新周期结束中断的，而当此中断发生时，本身就说明对CMOS RAM 的访问是安全的，毕竟留给我们的时间是999 毫秒，这段时间非常充裕，这段时间能执行千万条指令。所以，在这种特定的情况下，上面的判断过程是不必要的。当然，加上倒也无所谓。

第42～52 行，分别访问CMOS RAM 的0、2、4 号单元，从中读取当前的秒、分、时数据，按顺序压栈等待后续操作。

第60～62 行，读一下RTC 的寄存器C，使得所有中断标志复位。这等于是告诉RTC，中断已经得到处理，可以继续下一次中断。否则的话，RTC 看到中断未被处理，将不再产生中断信号。RTC 产生中断的原因有多种，可以在程序中通过读寄存器C 来判断具体的原因。不过这里不需要，因为除了更新周期结束中断外，其他中断都被关闭了。

现在，终于可以在屏幕上显示时间信息了。

第64、65 行，临时将段寄存器ES 指向显示缓冲区。

第67、68 行，首先从栈中弹出小时数，调用过程bcd_to_ascii 来将用BCD 码表示的“小时”转换成ASCII。该过程是在第105 行定义的，调用

该过程时，寄存器AL 中的高4 位和低4 位分别是“小时”的十位数字和个位数字。

第108 行，将寄存器AL 中的内容复制一份给AH，以方便下一步操作。

第109、110 行，将寄存器AL 中的高4 位清零，只留下“小时”的个位数字。接着，将它加上0x30，就得到该数字对应的ASCII 码。

十位上的数字在寄存器AH 的高4 位。第112 行，用右移4 位的方法，将它“拉”到低4位，高4 位在移动的过程中自动清零。

接着，第113、114 行，用同样的办法来得到十位数字的ASCII 码。此时，寄存器AH 中是十位数字的ASCII 码，AL 中是个位数字的ASCII 码，它们将作为结果返回给调用者。

最后，第116 行用于返回调用者。

接着回到第69 行，为了连续在屏幕上显示内容，最好是采用基址寻址来访问显存。这一行用于指定显示的内容位于显存的什么位置。实际上，这里指定的是第12 行36 列。同以前一样，每个字符在显存中占两个字节，每行80 个字符，所以这里使用了表达式12*160 + 36*2，该表达式的值是在编译阶段计算的。

第71、72 行，分别将“小时”的两个数位写到显存中，段地址在ES中，偏移地址分别是由寄存器BX 和BX+2 提供的。这里没有写入显示属性，这是因为我们希望采用默认的显示属性（屏幕是黑的，默认的显示属性是0x07，即黑底白字）。

第74、75 行，用于在下一个屏幕位置显示冒号“:”，这是在显示时间时都会采用的分隔符。当然，通过寄存器AL 中转是多余的，这两句可以直接写成

```
mov byte [es:bx+4],’:’
```

遗憾的是，等我发现这个问题时，本章已经快要写完了，重新排版实在太费工夫。其实，这不算是个问题，无伤大雅，难道不是吗？

为了验证RTC 更新结束中断是每秒发生一次的，第76 行，将冒号的显示属性（颜色）用not指令反转。就像手掌的两面一样，每次发生中断

时，冒号的颜色将和上一次相反，但永远在两个属性之间来回变化。到程序运行的时候你就会发现，变化的频率是每秒一次。

剩下的指令都很好理解，因为它们的工作是按相同的方法显示分钟数和秒数。第78～90 行，依次从栈中弹出分钟和秒的数值，并转换成ASCII 码，然后显示在屏幕上，中间用冒号间隔。

在8259 芯片内部，有一个中断服务寄存器（Interrupt Service Register，ISR），这是一个8 位寄存器，每一位都对应着一个中断输入引脚。当中断处理过程开始时，8259 芯片会将相应的位置1，表明正在服务从该引脚来的中断。

一旦响应了中断，8259 中断控制器无法知道该中断什么时候才能处理结束。同时，如果不清除相应的位，下次从同一个引脚出现的中断将得不到处理。在这种情况下，需要程序在中断处理过程的结尾，显式地对8259 芯片编程来清除该标志，方法是向8259 芯片发送中断结束命令（End Of Interrupt，EOI）。

中断结束命令的代码是0x20。代码清单9-1 第92～94 行就用来做这件事。需要注意的是，如果外部中断是8259 主片处理的，那么，EOI 命令仅发送给主片即可，端口号是0x20；如果外部中断是由从片处理的，就像本章的例子，那么，EOI 命令既要发往从片（端口号0xa0），也要发往主片。

最后，第96～102 行，从栈中恢复被中断程序的现场，并用中断返回指令iret 回到中断之前的地方继续执行。iret 的意思是Interrupt Return。

9.1.9 代码清单9-1 的编译和运行

本章的代码不包括加载器，也就是负责加载用户程序的主引导扇区代码，因为第8 章已经提供了一个加载器，它同样可以加载本章的用户程序。

在完全理解了代码清单9-1 的基础上，可以自行编辑和编译它，生成二进制文件。然后，使用FixVhdWr 工具将其写入虚拟硬盘。和第8 章一样，写入时的起始逻辑扇区号是100，毕竟加载器每次要从这个地方读取和加载用户程序。

一旦所有工作都准备停当，即可启动虚拟机来观察运行结果。通常情况下，运行结果会如图9-5 所示。

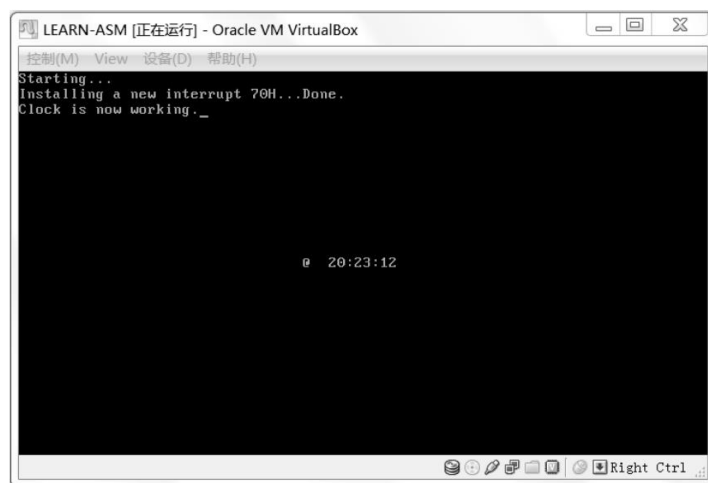


图9-5　代码清单9-1 编译和运行后的显示效果

在你欣赏程序的运行结果时，你一定会发现时间每秒更新一次，这从冒号的显示属性每秒反转一次可以看出来。与此不同的是，字符“@”却以很快的速度在闪烁。这意味着，把处理器从停机状态唤醒的不单单是实时时钟的更新周期结束中断，还有其他硬件中断，只不过我们不知道是谁而已。

9.2 内部中断

和硬件中断不同，内部中断发生在处理器内部，是由执行的指令引起的。比如，当处理器检测到div 或者idiv 指令的除数为零时，或者除法的结果溢出时，将产生中断0（0 号中断），这就是除法错中断。

再比如，当处理器遇到非法指令时，将产生中断6。非法指令是指指令的操作码没有定义，或者指令超过了规定的长度。操作码没有定义通常意味着那不是一条指令，而是普通的数。

内部中断不受标志寄存器IF 位的影响，也不需要中断识别总线周期，它们的中断类型是固定的，可以立即转入相应的处理过程。

9.3 软 中 断

软中断是由int 指令引起的中断处理。这类中断也不需要中断识别总线周期，中断号在指令中给出。int 指令的格式如下：

```
int3
int imm8
into
```

int3 是断点中断指令，机器指令码为CC。这条指令在调试程序的时候很有用，当程序运行不正常时，多数时候希望在某个地方设置一个检查点，也称断点，来查看寄存器、内存单元或者标志寄存器的内容，这条指令就是为这个目的而设的。

指令都是连续存放的，因此，所谓的断点，就是某条指令的起始地址。int3 是单字节指令，这是有意设计的。当需要设置断点时，可以将断点处那条指令的第1 字节改成0xcc，原字节予以保存。当处理器执行到int3 时，即发生3 号中断，转去执行相应的中断处理程序。中断处理程序的执行也要用到各个寄存器，这会破坏它们的内容，但push 指令不会。我们可以在该程序内先压栈所有相关寄存器和内存单元，然后分别取出予以显示，它们就是中断前的现场内容。最后，再恢复那条指令的第1 字节，并修改位于栈中的返回地址，执行iret 指令。

注意，int3 和int 3 不是一回事。前者的机器码为CC，后者则是CD 03，这就是通常所说的int n，其操作码为0xCD，第2 字节的操作数给出了中断号。举几个例子：

```
int 0x00          ;引发 0 号中断
int 0x15          ;引发 0x15 号中断
int 0x16          ;引发 0x16 号中断
```

into 是溢出中断指令，机器码为0xCE，也是单字节指令。当处理器执行这条指令时，如果标志寄存器的OF 位是1，那么，将产生4 号中断。否则，这条指令什么也不做。

9.3.1 BIOS 中断

可以为所有的中断类型自定义中断处理过程，包括内部中断、硬件中断和软中断。特别是考虑到处理器允许256 种中断类型，而且大部分都没有被硬件和处理器内部中断占用。

编写自己的中断处理程序有相当大的优越之处。不像jmp 和call 指令，int 指令不需要知道目标程序的入口地址。远转移指令jmp 和远调用指令call 必须直接或者间接给出目标位置的段地址和偏移地址，如果所有这一切都是自己安排的，倒也不成问题，但如果想调用别人的代码，比如操作系统的功能，这就很麻烦了。举个例子来说，假如你想读硬盘上的一个文件，因为操作系统有这样的功能，所以就不必在自己的程序中再写一套代码，直接调用操作系统例程就可以了。

但是，操作系统通常不会给出或者公布硬盘读写例程的段地址和偏移地址，因为操作系统也是经常修改的，经常发布新的版本。这样一来，例程的入口地址也会跟着变化。而且，也不能保证每次启动计算机之后，操作系统总待在同一个内存位置。

因为有了软中断，这是个利好条件。每次操作系统加载完自己之后，以中断处理程序的形式提供硬盘读写功能，并把该例程的地址填写到中断向量表中。这样，无论在什么时候，用户程序需要该功能时，直接发出一个软中断即可，不需要知道具体的地址。

最有名的软中断是BIOS 中断，之所以称为BIOS 中断，是因为这些中断功能是在计算机加电之后，BIOS 程序执行期间建立起来的。换句话说，这些中断功能在加载和执行主引导扇区之前，就已经可以使用了。

BIOS 中断，又称BIOS 功能调用，主要是为了方便地使用最基本的硬件访问功能。不同的硬件使用不同的中断号，比如，使用键盘服务时，中断号是0x16，即

```
int 0x16
```

通常，为了区分针对同一硬件的不同功能，使用寄存器AH 来指定具体的功能编号。举例来说，以下指令用于从键盘读取一个按键：

```
mov ah,0x00          ;从键盘读字符
int 0x16             ;键盘服务。返回时，字符代码在寄存器 AL 中
```

在这里，当寄存器AH 的内容是0x00 时，执行int 0x16 后，中断服务例程会监视键盘动作。当它返回时，会在寄存器AL 中存放按键的ASCII 码。

BIOS 中断很多，它们是在BIOS 执行期间安装的，当主引导程序开始执行时，就可以在程序中使用了。本准备给出一张BIOS 功能调用列表，但是考虑到现在网络技术很发达，上网很方便，大家可以自行从互联网上寻找相关的BIOS 功能调用资料，然后在自己的程序中做实验。

你可能觉得奇怪，BIOS 是怎么建立起这套功能调用中断的？它又是怎么知道如何访问硬件的？毕竟，即使是它，要访问硬件也得通过端口一级的途径。

答案是，BIOS 可能会为一些简单的外围设备提供初始化代码和功能调用代码，并填写中断向量表，但也有一些BIOS 中断是由外部设备接口自己建立的。

首先，每个外部设备接口，包括各种板卡，如网卡、显卡、键盘接口电路、硬件控制器等，都有自己的只读存储器（Read Only Memory，ROM），类似于BIOS 芯片，这些ROM 中提供了它自己的功能调用例程，以及本设备的初始化代码。按照规范，前两个单元的内容是0x55 和0xAA，第三个单元是本ROM 中以512 字节为单位的代码长度；从第四个单元开始，就是实际的ROM 代码。

其次，我们知道，从内存物理地址A0000 开始，到FFFFF 结束，有相当一部分空间是留给外围设备的。如果设备存在，那么，它自带的ROM 会映射到分配给它的地址范围内。

在计算机启动期间，BIOS 程序会以2KB 为单位搜索内存地址C0000～E0000 之间的区域。当它发现某个区域的头两个字节是0x55 和0xAA 时，那意味着该区域有ROM 代码存在，是有效的。接着，它对该区域做累加和检查，看结果是否和第三个单元相符。如果相符，就从第四个单元进入。这时，处理器执行的是硬件自带的程序指令，这些指令初始化外部设备的相关寄存器和工作状态，最后，填写相关的中断向量表，使它们指向自带的中断处理过程。

9.3.2　代码清单9-2

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：9-2（被加载的用户程序/BIOS 中断演示程序），源程序文件：c09_2.asm

9.3.3　从键盘读字符并显示

代码清单9-2 在框架上和前面的用户程序是一致的，差别在于代码段的功能上。

代码清单9-2 第28～32 行用于初始化各个段寄存器，这和以前的做法是相同的。

第34～42 行用于在屏幕上显示字符串，采用的是循环的方法。循环用的是loop 指令，为此，第34 行用于计算字符串的长度，并传送到寄存器CX 中，以控制循环的次数。第35 行用于取得字符串的首地址。

向屏幕上写字符使用的是BIOS 中断，具体地说就是中断0x10 的0x0e 号功能，该功能用于在屏幕上的光标位置处写一个字符，并推进光标位置。第38～40 行分别按规范的要求准备各个参数，执行软中断。

第41、42 行将递增寄存器BX 中的偏移地址，以指向下一个字符在数据段中的位置。然后，loop 指令将寄存器CX 的内容减1，并在其不为零的情况下返回到循环体开始处，继续显示下一个字符。

剩下的工作内容既复杂，又简单。复杂是指，从键盘读取你按下的那个键，并把它显示在屏幕上，很复杂，需要访问硬件，写一大堆指令。简单是指，因为有了BIOS 功能调用，这只需几条语句就能完成。

第45、46 行使用软中断0x16 从键盘读字符，需要在寄存器AH 中指定0x00 号功能。该中断返回后，寄存器AL 中为字符的ASCII 码。

第48～50 行又一次使用了int 0x10 的0x0e 号功能，把从键盘取得的字符显示在屏幕上。

第52 行，执行一个无条件转移指令，重新从键盘读取新的字符并予以显示。

9.3.4　代码清单9-2 的编译和运行

将代码清单9-2 编辑并编译后，用FixVhdWr 程序将生成的二进制文件写入虚拟硬盘，起始的逻辑扇区号同样为100。

如图9-6 所示，启动虚拟机后，会看到一段欢迎的话。现在，你可以按下任何按键，它们将原样显示在“->”之后。慢慢试验，细细体会，你会发现某些按键的特点。比如，回车键（Enter）仅仅是将光标移到行首，退格键（Backspace）仅仅是将光标退后，并不破坏该位置上的字符。

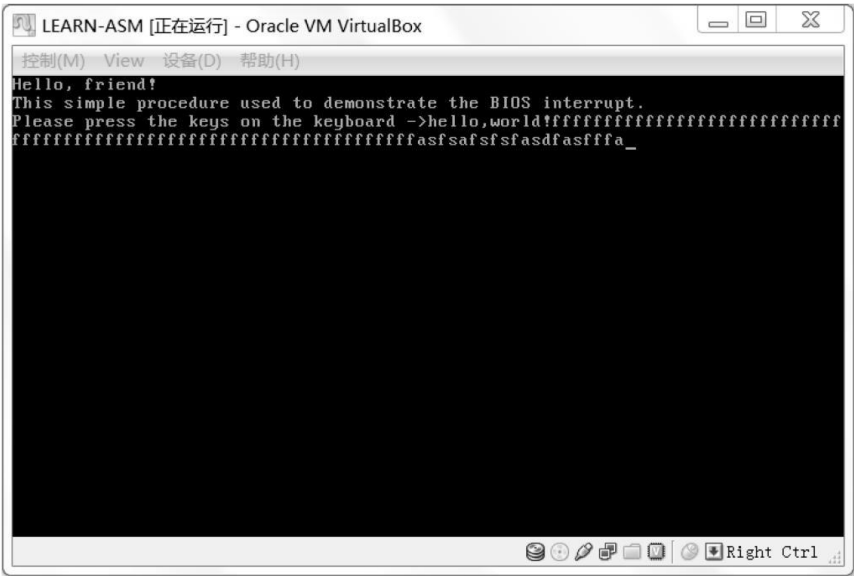


图9-6　代码清单9-2 编译并运行后的效果

本 章 习 题

1．修改代码9-1，对8259 芯片编程，屏蔽除RTC 外的其他所有中断，观察字符“@”的变化速度。

2．修改代码9-1，使之用一种新的方法来产生中断信号。建议的方法是采用周期性中断。不过，这涉及选择分频电路的分节点，比如，你可以选择250ms 或者500ms，它们分别会在1 秒种内产生4 次或2 次中断。

第3部分　32位保护模式

学习处理器32位保护模式的工作原理，包括分段、分页、特权级、保护、中断和异常中断等。

学习32位保护模式下的汇编语言程序设计技术。

通过多个实例了解操作系统如何在保护模式下加载应用程序，并提供各种管理服务。

学会用Bochs虚拟机调试32位保护模式下的程序。

第10章　32 位x86 处理器编程架构

所谓处理器架构，或者处理器编程架构，是指一整套的硬件结构，以及与之相适应的工作状态，这其中的灵魂部分就是一种设计理念，决定了处理器的应用环境和工作模式，也决定了软件开发人员如何在这种模式下解决实际问题。架构内的资源对程序员来说是可见的、可访问的，受程序的控制以改变处理器的运行状态；非架构的资源取决于具体的硬件实现。

处理器架构实际上是不断扩展的，新处理器必须延续旧的设计思路，并保持兼容性和一致性；同时还会有所扩充和增强。

Intel 32 位处理器架构简称IA-32（Intel Architecture，32-bit），是以1978 年的8086 处理器为基础发展起来的。在那个时候，他们只是想造一款特别牛的处理器，也没考虑到架构。尽管那些人是专家，但和我们一样不是千里眼，这是很正常的。

正如我们已经知道的，8086 有20 根地址线，可以寻址1MB 内存。但是，它内部的寄存器是16 位的，无法在程序中访问整个1MB 内存。所以，它也是第一款支持内存分段模型的处理器。还有，8086 处理器只有一种工作模式，即实模式。当然，在那时，还没有实模式这一说。

由于8086 处理器的成功，推动着Intel 公司不断地研发更新的处理器，32 位的时代就这样到来了。到目前为止，到底有多少种类型，我也说不清楚。尽管8086 是16 位的处理器，但它也是32 位架构内的一部分。原因在于，32 位的处理器架构是从8086 那里发展来的，是基于8086 的，具有延续性和兼容性。

就我们曾经用过的产品而言，32 位的处理器有32 根地址线，数据线的数量是32 根或者64根。特别是最近最新的处理器，都是64 根。因此，它可以访问232，即4GB 的内存，而且每次可以读写连续的4 字节或者8 字节，这称为双字（Double Word）或者4 字（Quad Word）访问。当然，如果你要按字节或者字来访问内存，也是允许的。

我总说，处理器虽小，功能却异常复杂。要想把32 位处理器的所有功能都解释清楚，不是一件简单的事情。它不单单是地址线和数据线的

扩展，实际上还有更多的部分，包括高速缓存、流水线、浮点处理部件、多处理器（核）管理、多媒体扩展、乱序执行、分支预测、虚拟化、温度和电源管理等。在这本书里，我的一个基本原则是，如果你不能讲清楚，干脆就不要提它。因此，我只讲那些现在用得上的东西。

10.1　IA-32 架构的基本执行环境

10.1.1　寄存器的扩展

在16 位处理器内，有8 个通用寄存器AX、BX、CX、DX、SI、DI、BP 和SP，其中，前4个还可以拆分成两个独立的8 位寄存器来用，即AH、AL、BH、BL、CH、CL、DH 和DL。如图10-1 所示，32 位处理器在16 位处理器的基础上，扩展了这8 个通用寄存器的长度，使之达到32位。

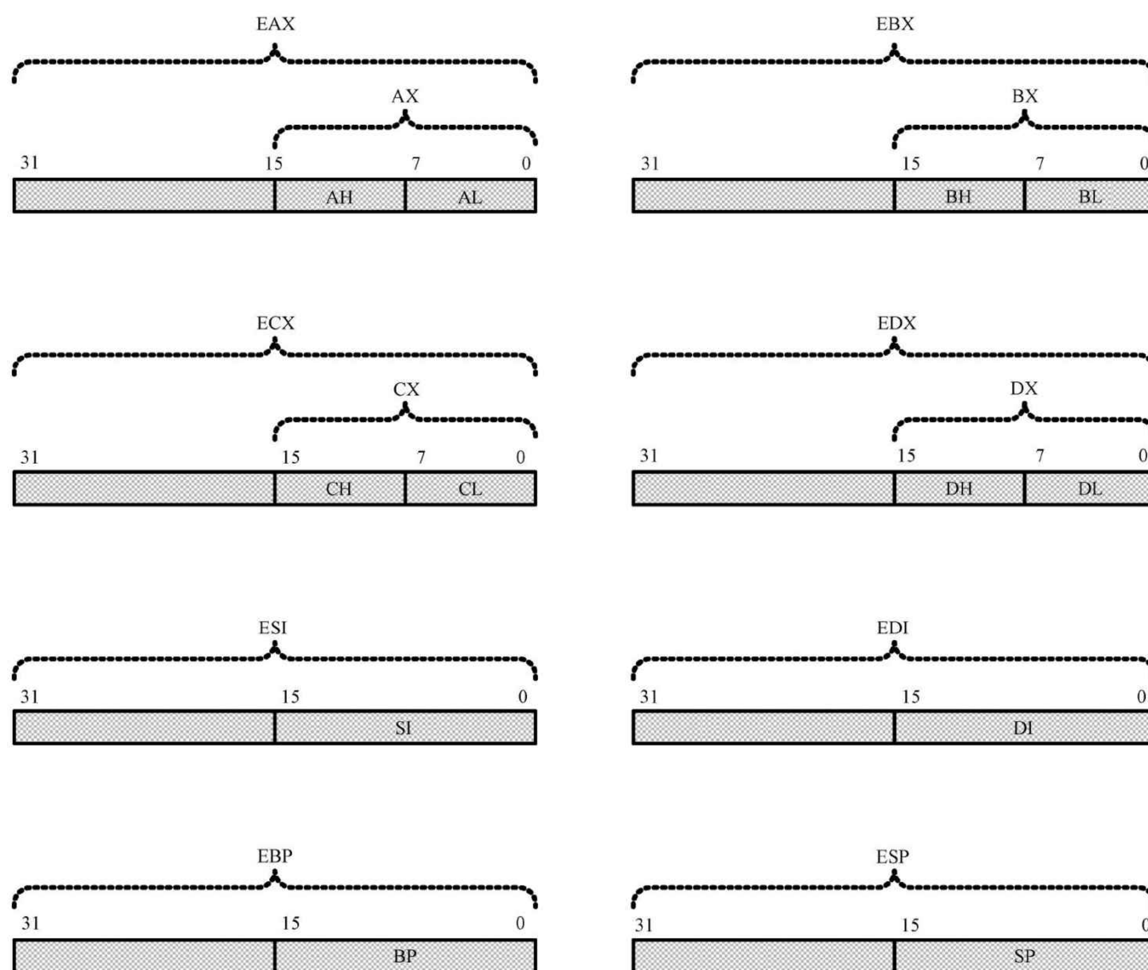


图10-1　32 位处理器内部的通用寄存器

为了在汇编语言程序中使用经过扩展（Extend）的寄存器，需要给它们命名，它们的名字分别是EAX、EBX、ECX、EDX、ESI、EDI、ESP 和EBP。可以在程序中使用这些寄存器，即使是在实模式下：

```
mov eax,0xf0000005
mov ecx,eax
add edx,ecx
```

但是，就像以上指令所示的那样，指令的源操作数和目的操作数必须具有相同的长度，个别特殊用途的指令除外。因此，像这样的搭配是不允许的，在程序编译时，编译器会报告错误：

```
mov eax,cx            ;错误的汇编语言指令
```

如果目的操作数是32 位寄存器，源操作数是立即数，那么，立即数被视为32 位的：

```
mov eax,0xf5          ;EAX←0x000000f5
```

32 位通用寄存器的高16 位是不可独立使用的，但低16 位保持同16 位处理器的兼容性。因此，在任何时候它们都可以照往常一样使用：

```
mov ah,0x02
mov al,0x03
add ax,si
```

可以在32 位处理器上运行16 位处理器上的软件。但是，它并不是16 位处理器的简单增强。事实上，32 位处理器有自己的32 位工作模式，在本书中，32 位模式特指32 位保护模式。在这种模式下，可以完全、充分地发挥处理器的性能。同时，在这种模式下，处理器可以使用它全部的32 根地址线，能够访问4GB 内存。

如图10-2 所示，在32 位模式下，为了生成32 位物理地址，处理器需要使用32 位的指令指针寄存器。为此，32 位处理器扩展了IP，使之达到32 位，即EIP。当它工作在16 位模式下时，依然使用16 位的IP；工作在32 位模式下时，使用的是全部的32 位EIP。和往常一样，即使是在32 位模式下，EIP 寄存器也只由处理器内部使用，程序中是无法直接访问

的。对IP 和EIP 的修改通常是用某些指令隐式进行的，这些指令包括JMP、CALL、RET 和IRET 等等。

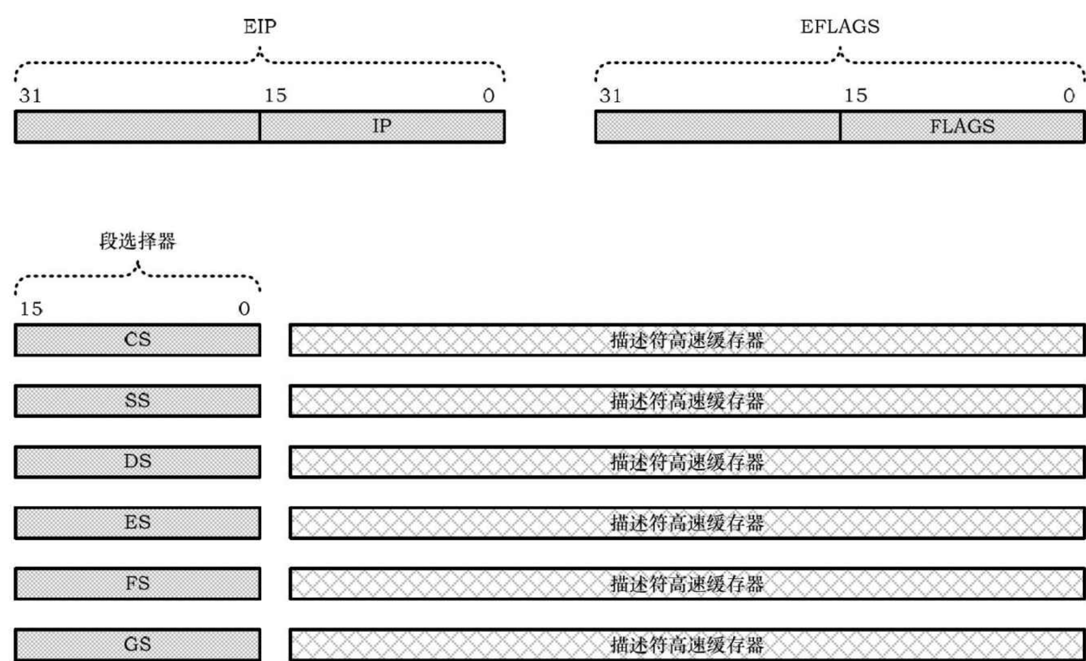


图10-2　32 位处理器的指令指针、标志和段寄存器

另外，在16 位处理器中，标志寄存器FLAGS 是16 位的，在32 位处理器中，扩展到了32位，低16 位和原先保持一致。关于EFLAGS 中的各个标志位，将在后面的章节中逐一介绍。

在32 位模式下，对内存的访问从理论上来说不再需要分段，因为它有32 根地址线，可以自由访问任何一个内存位置。但是，IA-32 架构的处理器是基于分段模型的，因此，32 位处理器依然需要以段为单位访问内存，即使它工作在32 位模式下。

不过，它也提供了一种变通的方案，即，只分一个段，段的基地址是0x00000000，段的长度（大小）是4GB。在这种情况下，可以视为不分段，即平坦模型（Flat Mode）。

每个程序都有属于自己的内存空间。在16 位模式下，一个程序可以自由地访问不属于它的内存位置，甚至可以对那些地方的内容进行修改。这当然是不安全的，也不合法，但却没有任何机制来限制这种行为。在32 位模式下，处理器要求在加载程序时，先定义该程序所拥有的段，然后允许使用这些段。定义段时，除了基地址（起始地址）外，还

附加了段界限、特权级别、类型等属性。当程序访问一个段时，处理器将用固件实施各种检查工作，以防止对内存的违规访问。

如图10-2 所示，在32 位模式下，传统的段寄存器，如CS、SS、DS、ES，保存的不再是16位段基地址，而是段的选择子，即，用于选择所要访问的段，因此，严格地说，它的新名字叫做段选择器。除了段选择器之外，每个段寄存器还包括一个不可见部分，称为描述符高速缓存器，里面有段的基地址和各种访问属性。这部分内容程序不可访问，由处理器自动使用。

有关32 位模式下的段和段的访问方法，将在后面的章节中予以详述，你在看这段文字的时候，也许有迷迷糊糊的感觉，没关系，这是正常的，到后面你就会感觉豁然开朗了。

最后，32 位处理器增加了两个额外的段寄存器FS 和GS。对于某些复杂的程序来说，多出两个段寄存器可能会令它们感到高兴。

10.1.2　基本的工作模式

8086 具有16 位的段寄存器、指令指针寄存器和通用寄存器（CS、SS、DS、ES、IP、AX、BX、CX、DX、SI、DI、BP、SP），因此，我们称它为16 位的处理器。尽管它可以访问1MB 的内存，但是只能分段进行，而且由于只能使用16 位的段内偏移量，故段的长度最大只能是64KB。8086 只有一种工作模式，即实模式。当然，这个名称是后来才提出来的。

1982 年的时候，Intel 公司推出了80286 处理器。这也是一款16 位的处理器，大部分的寄存器都和8086 处理器一样。因此，80286 和8086一样，因为段寄存器是16 位的，而且只能使用16位的偏移地址，在实模式下只能使用64KB 的段；尽管它有24 根地址线，理论上可以访问224，即16MB 的内存,但依然只能分成多个段来进行。

但是，80286 和8086 不一样的地方在于，它第一次提出了保护模式的概念。在保护模式下，段寄存器中保存的不再是段地址，而是段选择子，真正的段地址位于段寄存器的描述符高速缓存中，是24 位的。因此，运行在保护模式下的80286 处理器可以访问全部16MB 内存。

80286 处理器访问内存时，不再需要将段地址左移，因为在段寄存器的描述符高速缓存器中有24 位的段物理基地址。这样一来，段可以位

于16MB 内存空间中的任何位置，而不再限于低端1MB 范围内，也不必非得是位于16 字节对齐的地方。不过，由于80286 的通用寄存器是16位的，只能提供16 位的偏移地址，因此，和8086 一样，即使是运行在保护模式下，段的长度依然不能超过64KB。对段长度的限制妨碍了80286 处理器的应用，这就是16 位保护模式很少为人所知的原因。

实模式等同于8086 模式，在本书中，实模式和16 位保护模式统称16 位模式。在16 位模式下，数据的大小是8 位或者16 位的；控制转移和内存访问时，偏移量也是16 位的。

1985 年的80386 处理器是Intel 公司的第一款32 位产品，而且获得了极大成功，是后续所有32 位产品的基础。本书中的绝大多数例子，都可以在80386 上运行。和8086/80286 不同，80386处理器的寄存器是32位的，而且拥有32 根地址线，可以访问232，即4GB 的内存。

80386，以及所有后续的32 位处理器，都兼容实模式，可以运行实模式下的8086 程序。而且，在刚加电时，这些处理器都自动处于实模式下，此时，它相当于一个非常快速的8086 处理器。只有在进行一番设置之后，才能运行在保护模式下。

在保护模式下，所有的32 位处理器都可以访问多达4GB 的内存，它们可以工作在分段模型下，每个段的基地址是32 位的，段内偏移量也是32 位的，因此，段的长度不受限制。在最典型的情况下，可以将整个4GB 内存定义成一个段来处理，这就是所谓的平坦模式。在平坦模式下，可以执行4GB 范围内的控制转移，也可以使用32 位的偏移量访问任何4GB 范围内的任何位置。32 位保护模式兼容80286 的16 位保护模式。

除了保护模式，32 位处理器还提供虚拟8086 模式（V86 模式），在这种模式下，IA-32 处理器被模拟成多个8086 处理器并行工作。V86 模式是保护模式的一种，可以在保护模式下执行多个8086 程序。传统上，要执行8086 程序，处理器必须工作在实模式下。在这种情况下，为32 位保护模式写的程序就不能运行。但是，V86 模式提供了让它们在一起同时运行的条件。

V86 模式曾经很有用，因为在那个时候，8086 程序很多，而32 位应用程序很少，这个过渡期是必需的。现在，这种工作模式已经基本无用了。

在本书中，32 位模式特指IA-32 处理器上的32 位保护模式。不存在所谓的32 位实模式，实模式的概念实质上就是8086 模式。

10.1.3 线性地址

为IA-32 处理器编程，访问内存时，需要在程序中给出段地址和偏移量，因为分段是IA-32 架构的基本特征之一。传统上，段地址和偏移地址称为逻辑地址，偏移地址叫做有效地址（Effective Address，EA），在指令中给出有效地址的方式叫做寻址方式（Addressing Mode）。比如：

```
inc word [bx+si+0x06]
```

在这里，指令中使用的是基址加变址的方式来寻找最终的操作数。

段的管理是由处理器的段部件负责进行的，段部件将段地址和偏移地址相加，得到访问内存的地址。一般来说，段部件产生的地址就是物理地址。

IA-32 处理器支持多任务。在多任务环境下，任务的创建需要分配内存空间；当任务终止后，还要回收它所占用的内存空间。在分段模型下，内存的分配是不定长的，程序大时，就分配一大块内存；程序小时，就分配一小块。时间长了，内存空间就会碎片化，就有可能出现一种情况：内存空间是有的，但都是小块，无法分配给某个任务。为了解决这个问题，IA-32 处理器支持分页功能，分页功能将物理内存空间划分成逻辑上的页。页的大小是固定的，一般为4KB，通过使用页，可以简化内存管理。

如图10-3 所示，当页功能开启时，段部件产生的地址就不再是物理地址了，而是线性地址（Linear Address），线性地址还要经页部件转换后，才是物理地址。

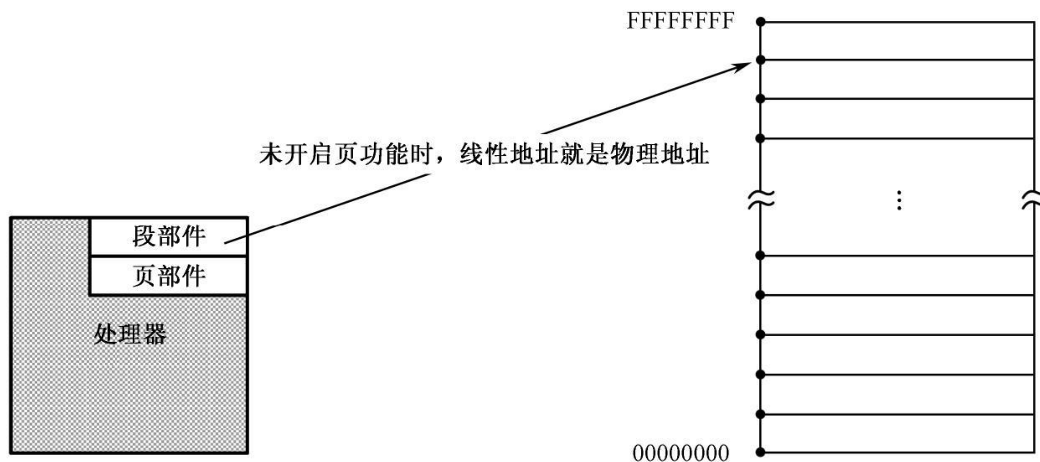


图10-3　线性地址和线性地址空间

线性地址的概念用来描述任务的地址空间。如图10-3 所示，IA-32 处理器上的每个任务都拥有4GB 的虚拟内存空间，这是一段长4GB 的平坦空间，就像一段平直的线段，因此叫线性地址空间。相应地，由段部件产生的地址，就对应着线性地址空间上的每一个点，这就是线性地址。

IA-32 架构下的任务、分段、分页等内容，是本书的重点，要在后半部分详细论述。

10.2　现代处理器的结构和特点

10.2.1　流水线

处理器的每一次更新换代，都会增加若干新特性，这是很自然的。同时我们也会发现，老软件在新的处理器上跑得更快。这里面的原因很简单，处理器的设计者总是在想尽办法加快指令的执行。

早在8086 时代，处理器就已经有了指令预取队列。当指令执行时，如果总线是空闲的（没有访问内存的操作），就可以在指令执行的同时预取指令并提前译码，这种做法是有效的，能大大加快程序的执行速度。

处理器可以做很多事情，换言之，能够执行各种不同的指令，完成不同的功能，但这些事情大都不会在一个时钟周期内完成。执行一条指令需要从内存中取指令、译码、访问操作数和结果，并进行移位、加法、减法、乘法以及其他任何需要的操作。

为了提高处理器的执行效率和速度，可以把一条指令的执行过程分解成若干个细小的步骤，并分配给相应的单元来完成。各个单元的执行是独立的、并行的。如此一来，各个步骤的执行在时间上就会重叠起来，这种执行指令的方法就是流水线（Pipe-Line）技术。

比如，一条指令的执行过程分为取指令、译码和执行三个步骤，而且假定每个步骤都要花1个时钟周期，那么，如图10-4 所示，如果采用顺序执行，则执行三条指令就要花9 个时钟周期，每3 个时钟周期才能得到一条指令的执行结果；如果采用3 级流水线，则执行这三条指令只需5个时钟周期，每隔一个时钟周期就能得到一条指令的执行结果。

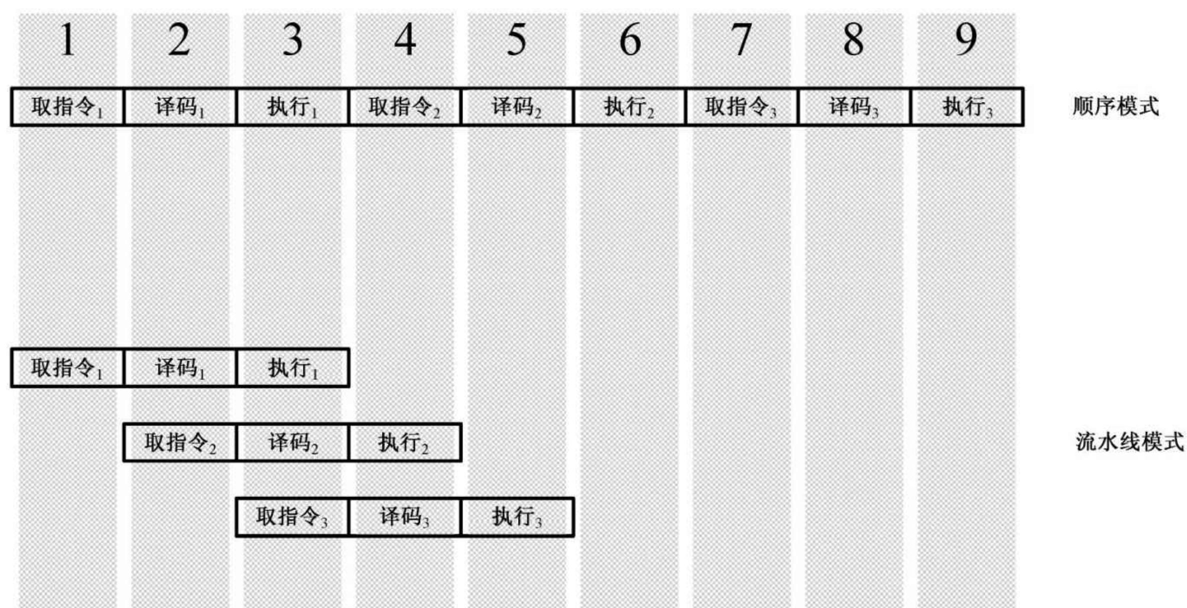


图10-4　流水线的基本原理

一个简单的流水线其实不过如此，但是，它仍有很大的改进空间。原因很简单，指令的执行过程仍然可以继续细分。一般来说，流水线的效率受执行时间最长的那一级的限制，要缩短各级的执行时间，就必须让每一级的任务减少，与此同时，就需要把一些复杂的任务再进行分解。比如，2000 年之后推出的Pentium 4 处理器采用了NetBurst 微结构，它进一步分解指令的执行过程，采用了31 级超深流水线。

10.2.2　高速缓存

影响处理器速度的另一个因素是存储器。从处理器内部向外看，它们分别是寄存器、内存和硬盘。当然，现在有的计算机已经用上了固态磁盘。

寄存器的速度是最快的，原因在于它使用了触发器，这是一种利用反馈原理制作的存储电路，在《穿越计算机的迷雾》那本书里，介绍得很清楚。触发器的工作速度是纳秒（ns）级别的，当然也可以用来作为内存的基本单元，即静态存储器（SRAM），缺点是成本太高，价格也不菲。所以，制作内存芯片的材料一般是电容和单个的晶体管，由于电容需要定时刷新，使得它的访问速度变得很慢，通常是几十个纳秒。因此，它也获得了一个恰当的名字：动态存储器（DRAM），我们所用的

内存芯片，大部分都是DRAM。最后，硬盘是机电设备，是机械和电子的混合体，它的速度最慢，通常在毫秒级（ms）。

在这种情况下，因为需要等待内存和硬盘这样的慢速设备，处理器便无法全速运行。为了缓解这一矛盾，高速缓存（Cache）技术应运而生。高速缓存是处理器与内存（DRAM）之间的一个静态存储器，容量较小，但速度可以与处理器匹配。

高速缓存的用处源于程序在运行时所具有的局部性规律。首先，程序常常访问最近刚刚访问过的指令和数据，或者与它们相邻的指令和数据。比如，程序往往是序列化地从内存中取指令执行的，循环操作往往是执行一段固定的指令。当访问数据时，要访问的数据通常都被安排在一起；其次，一旦访问了某个数据，那么，不久之后，它可能会被再次访问。

利用程序运行时的局部性原理，可以把处理器正在访问和即将访问的指令和数据块从内存调入高速缓存中。于是，每当处理器要访问内存时，首先检索高速缓存。如果要访问的内容已经在高速缓存中，那么，很好，可以用极快的速度直接从高速缓存中取得，这称为命中（Hit）；否则，称为不中（miss）。在不中的情况下，处理器在取得需要的内容之前必须重新装载高速缓存，而不只是直接到内存中去取那个内容。高速缓存的装载是以块为单位的，包括那个所需数据的邻近内容。为此，需要额外的时间来等待块从内存载入高速缓存，在该过程中所损失的时间称为不中惩罚（miss penalty）。

高速缓存的复杂性在于，每一款处理器可能都有不同的实现。在一些复杂的处理器内部，会存在多级Cache，分别应用于各个独立的执行部件。

10.2.3　乱序执行

为了实现流水线技术，需要将指令拆分成更小的可独立执行部分，即拆分成微操作（microoperations），简写为μops。

有些指令非常简单，因此只需要一个微操作。如：

```
add eax,ebx
```

再比如：

```
add  eax,[mem]
```

可以拆分成两个微操作，一个用于从内存中读取数据并保存到临时寄存器，另一个用于将EAX 寄存器和临时寄存器中的数值相加。

再举个例子，这条指令：

```
add [mem],eax
```

可以拆分成三个微操作，一个从内存中读数据，一个执行相加的动作，第3 个用于将相加的结果写回到内存中。

一旦将指令拆分成微操作，处理器就可以在必要的时候乱序执行（Out-Of-Order Execution）程序。考虑以下例子：

```
mov eax,[mem1]
shl eax,5
add eax,[mem2]
mov [mem3],eax
```

这里，指令add eax,[mem2]可以拆分为两个微操作。如此一来，在执行逻辑左移指令的同时，处理器可以提前从内存中读取mem2 的内容。典型地，如果数据不在高速缓存中（不中），那么处理器在获取mem1 的内容之后，会立即开始获取mem2 的内容，与此同时，shl 指令的执行早就开始了。

将指令拆分成微操作，也可以使得栈的操作更有效率。考虑以下代码片断：

```
push eax
call func
```

这里，push eax 指令可以拆分成两个微操作，即可以表述为以下的等价形式：

```
sub esp,4
mov [esp],eax
```

这就带来了一个好处，即使EAX 寄存器的内容还没有准备好，微操作sub esp,4 也可以执行。call 指令执行时需要在当前栈中保存返回地址，在以前，该操作只能等待push eax 指令执行结束，因为它需要ESP 的新值。感谢微操作，现在，call 指令在微操作sub esp,4 执行结束时就可以无延迟地立即开始执行。

10.2.4 寄存器重命名

考虑以下例子：

```
mov eax,[mem1]
shl eax,3
mov [mem2],eax
mov eax,[mem3]
add eax,2
mov [mem4],eax
```

以上代码片断做了两件事，但互不相干：将mem1 里的内容左移3 次（乘以8），并将mem3里的内容加2。如果我们为最后三条指令使用不同的寄存器，那么将更明显地看出这两件事的无关性。并且，事实上，处理器实际上也是这样做的。处理器为最后三条指令使用了另一个不同的临时寄存器，因此，左移（乘法）和加法可以并行地处理。

IA-32 架构的处理器只有8 个32 位通用寄存器，但通常都会被我们全部派上用场（甚至还觉得不够）。因此，我们不能奢望在每个计算当中都使用新的寄存器。不过，在处理器内部，却有大量的临时寄存器可用，处理器可以重命名这些寄存器以代表一个逻辑寄存器，比如EAX。

寄存器重命名以一种完全自动和非常简单的方式工作。每当指令写逻辑寄存器时，处理器就为那个逻辑寄存器分配一个新的临时寄存器。再来看一个例子：

```
mov eax,[mem1]
mov ebx,[mem2]
add ebx,eax
shl eax,3
mov [mem3],eax
mov [mem4],ebx
```

假定现在mem1 的内容在高速缓存里，可以立即取得，但mem2 的内容不在高速缓存中。这意味着，左移操作可以在加法之前开始（使用临时寄存器代替EAX）。为左移的结果使用一个新的临时寄存器，其好处是EAX 寄存器中仍然是以前的内容，它将一直保持这个值，直到EBX 寄存器中的内容就绪，然后同它一起做加法运算。如果没有寄存器重命名机制，左移操作将不得不等待从内存中读取mem2 的内容到EBX 寄存器以及加法操作完成。

在所有的操作都完成之后，那个代表EAX 寄存器最终结果的临时寄存器的内容被写入真实的EAX 寄存器，该处理过程称为引退（Retirement）。

所有通用寄存器，栈指针、标志、浮点寄存器，甚至段寄存器都有可能被重命名。

10.2.5　分支目标预测

流水线并不是百分之百完美的解决方案。实际上，有很多潜在的因素会使得流水线不能达到最佳的效率。一个典型的情况是，如果遇到一条转移指令，则后面那些已经进入流水线的指令就都无效了。换句话说，我们必须清空（Flush）流水线，从要转移到的目标位置处重新取指令放入流水线。

在现代处理器中，流水线操作分为很多步骤，包括取指令、译码、寄存器分配和重命名、微操作排序、执行和引退。指令的流水线处理方式允许处理器同时做很多事情。在一条指令执行时，下一条指令正在获取和译码。

流水线的最大问题是代码中经常存在分支。举个例子来说，一个条件转移允许指令流前往任意两个方向。如果这里只有一个流水线，那么，直到那个分支开始执行，在此之前，处理器将不知道应该用哪个分支填充流水线。流水线越长，处理器在用错误的分支填充流水线时，浪费的时间越多。

随着复杂架构下的流水线变得越来越长，程序分支带来的问题开始变得很大。让处理器的设计者不能接受，毕竟不中处罚的代价越来越高。

为了解决这个问题，在1996 年的Pentium Pro 处理器上，引入了分支预测技术（Branch Prediction）。分支预测的核心问题是，转移是发生还是不会发生。换句话说，条件转移指令的条件会不会成立。举个例子来说：

```
jne branch5
```

在这条指令还没有执行的时候，处理器就必须提前预测相等的条件在这条指令执行的时候是否成立。这当然是很困难的，几乎不可能。想想看，如果能够提前知道结果，还执行这些指令干嘛。

但是，从统计学的角度来看，有些事情一旦出现，下一次还会出现的概率较大。一个典型的例子就是循环，比如下面的程序片断：

```
    xor si,si
lops:
```

```
    ......
    cmp si,20
    jnz lops
```

当jnz 指令第一次执行时，转移一定会发生。那么，处理器就可以预测，下一次它还会转移到标号lops 处，而不是顺序往下执行。事实上，这个预测通常是很准的。

在处理器内部，有一个小容量的高速缓存器，叫分支目标缓存器（Branch Target Buffer，BTB）。当处理器执行了一条分支语句后，它会在BTB 中记录当前指令的地址、分支目标的地址，以及本次分支预测的结果。下一次，在那条转移指令实际执行前，处理器会查找BTB，看有没有最近的转移记录。如果能找到对应的条目，则推测执行和上一次相同的分支，把该分支的指令送入流水线。

当该指令实际执行时，如果预测是失败的，那么，清空流水线，同时刷新BTB 中的记录。这个代价较大。

10.3　32 位模式的指令系统

10.3.1　32 位处理器的寻址方式

在16 位处理器上，指令中的操作数可以是8 位或者16 位的寄存器、指向8 位或者16 位实际操作数的16 位内存地址，以及8 位或16 位的立即数。

如果指令中包含了内存地址操作数，那么，它必然是一个16 位的段内偏移地址，称为有效地址。通过有效地址，可以间接取得8 位或者16 位的实际操作数。指定有效地址可以使用基址寄存器BX、BP，变址（索引）寄存器SI 和DI，同时还可以加上一个8 位或16 位的偏移量。比如：

```
mov ax,[bx]
mov ax,[bx+di]
mov al,[bx+si+0x02]
```

以上，第1 条指令，寄存器BX 中的内容是指向16 位实际操作数的16 位地址；第2 条指令，寄存器BX 和DI 的内容相加，形成指向16 位实际操作数的16 位地址；第3 条指令，寄存器BX、SI 和8 位偏移量共同形成指向8 位实际操作数的16 位地址。

如图10-5 所示，这是16 位处理器的内存寻址方式示意图。从图中可以看出，允许使用基址寄存器BX 或者BP，同变址寄存器SI 或者DI 结合，再加上8 位或者16 位偏移量来寻址内存操作数。

图10-5　16 位处理器的内存寻址方式

16 位处理器的寻址方式本来就很复杂，当32 位处理器出现后，寄存器和偏移地址的宽度都扩展了，相应地，要继续扩展原有的寻址方式。但是，原有的16 位方案已经成型，再进行修补是非常困难的。一个可行

的解决方案是，让16 位指令和32 位指令共用相同的指令码，但通过不同的指令前缀，结合处理器当前的运行状态来决定该指令的寻址方式。

比如，当处理器运行在16 位模式时，如果没有指令前缀0x66，则认为指令是传统的16 位寻址方式；若有指令前缀0x66，则指令是新的32 位寻址方式。如果处理器当前运行在32 位模式下且没有指令前缀0x66，则视为默认的32 位寻址方式，否则就是传统的16 位寻址方式。

32 位处理器兼容16 位处理器的工作模式，可以运行传统的16 位代码。但是，它有自己独立的32 位运行模式，而且只有在这种模式下才能发挥最高的运行效率。

在32 位模式下，默认使用32 位宽度的寄存器。如：

```
mov eax,ebx
```

如果指令中使用了立即数，那么，该数值默认是32 位的：

```
mov ecx,0x55      ;ECX←0x00000055
```

还有，如果指令中的操作数是指向内存单元的地址，那么，该地址默认是32 位的段内偏移地址，或者叫段内偏移量：

```
mov edx,[mem]     ;mem 是一个 32 位的段内偏移地址
```

这就是说，如果指令中包含了内存地址操作数，那么，它必然默认地是一个32 位的有效地址。通过有效地址，可以间接取得32 位的实际操作数。如图10-6 所示，指定有效地址可以使用全部的32 位通用寄存器作为基址寄存器。同时，还可以再加上一个除ESP 之外的32 位通用寄存器作为变址寄存器。变址寄存器还允许乘以1、2、4 或者8 作为比例因子。最后，还允许加上一个8 位或者32 位的偏移量。

(EAX EBX ECX EDX ESP EBP ESI EDI) + (EAX EBX ECX EDX EBP ESI EDI) × (1 2 4 8) + (8位或32位偏移量)

图10-6　32 位处理器的内存寻址方式

以下是几个例子：

```
add eax,[0x2008]            ;有效地址为 0x00002008
sub eax,[eax+0x08]          ;有效地址是 32 位的
mov ecx,[eax+ebx*8+0x02]    ;有效地址是 32 位的
```

值得说明的是，在16 位模式下，内存寻址方式的操作数不允许使用栈指针寄存器SP。因此，象这条指令就是不正确的：

```
mov ax,[sp]
```

但是，在32 位模式下，允许在内存操作数中使用栈指针寄存器ESP。因此，下面的指令形式是合法的：

```
mov eax,[esp]
```

10.3.2 操作数大小的指令前缀

Intel 处理器的指令系统比较复杂，这种复杂性来源于两个方面，一是指令的数量较多，二是寻址方式也很多。可以想象，为了组成这些众多的指令，必须有一套同样复杂的指令格式。

如图10-7 所示，每一条处理器指令都可以拥有前缀，比如重复前缀（REP/REPE/ REPNE）、段超越前缀（如ES：）、总线封锁前缀（LOCK）等。前缀是可选的，每个前缀的长度是1 字节，每条指令可以有1～4 个前缀，或者不使用前缀。

前缀（如果有的话）的后面是操作码部分，指示执行什么样的操作，比如传送、加法、减法、乘法、除法、移位等。根据指令的不同，操作码的长度是1～3 字节。同时，操作码还可以用来指示操作的字长，即数据宽度为字节还是字。

操作码之后是操作数类型和寻址方式部分。这部分是可选的，简单的指令不包含这一部分，稍微复杂一点的指令，这一部分只有1 字节；最复杂的指令，可能有2 字节。这部分给出了指令的寻址方式，以及寄存器的类型（用的是哪个寄存器）。

指令的最后是立即数和偏移量。如果指令中使用了立即数，那么立即数就在这一部分给出；如果指令使用了带偏移量的寻址方式，如：

```
mov cx,[0x2000]
mov ecx,[eax+ebx*8+0x02]
```

那么，偏移量0x2000 和0x02 也在这部分出现。取决于具体的指令，立即数可以是1、2 或者4 字节，偏移量部分与此相同。

前缀	操作码	寻址方式和操作数类型	立即数	偏移量

图10-7　IA-32 的指令格式

上述的指令编码格式发源于16 位处理器时代，并在32 位处理器出现之后做了修改，主要是扩展了数据的宽度，其他都保持不变。毕竟，兼容性是首要考虑的因素。但是，这也带来了一些问题。考虑以下指令：

```
mov dx,[bx+si+0x02]
```

在16 位指令编码格式中，这种内存单元到寄存器的传送指令使用了操作码0x8B。如图10-8（a）所示，在操作码0x8B 之后是1 字节的寻址方式和操作数类型部分。位7 和位6 的值是01，表示使用了基地址变址的寻址方式，而且带有8 位偏移量；位5～位3 的值是010，指示目的操作数为寄存器DX；位2～位0 的值是000，表示寻址方式为“BX+SI+8 位偏移量”。在该字节之后，是1 字节的偏移量0x02。因此，这条指令编译后的机器代码是

```
8B 50 02
```

32 位处理器使用相同的编码格式，但是，寻址方式和寄存器的定义却是另起炉灶的，完全不同于16 位指令。如图10-8（b）所示，在32 位处理器上，位7 和位6 的值是01，表示使用了基址寻址方式，而且带有8 位偏移量；位5～位3 的值是010，指示目的操作数为寄存器EDX；位2～位0 的值是000，表示寻址方式为EAX+8 位偏移量。在该字节之后，是1 字节的偏移量0x02。因此，同样的机器指令码，却对应着不同的32 位指令：

```
mov edx,[eax+0x02]
```

这就是说，相同的机器指令，在16 位模式下和32 位模式下的解释和执行效果是不同的。但是，别忘了，32 位处理器可以执行16 位的程序，包括实模式和16 位保护模式。为此，在16 位模式下，处理器把所有指令都看成是16 位的。举个例子，机器指令码0x40 在16 位模式下的含义是

```
inc ax
```

当处理器在16 位模式下运行时，也可以使用32 位的寄存器，执行32 位的运算。为此，必须使用指令前缀0x66 来临时改变这种默认状态，因为同一个指令码，在16 位模式下和32 位模式下具有不同的解释。因此，当处理器在16 位模式下运行时，机器指令码

```
66 40
```

对应的指令不再是inc ax，而是

```
inc eax
```

相反地，如果处理器运行在32 位模式下，那么，处理器认为指令的操作数都是32 位的，如果你加了前缀，这个前缀就用来指示指令是16 位的。因此，指令前缀0x66 具有反转当前默认操作数大小的作用。

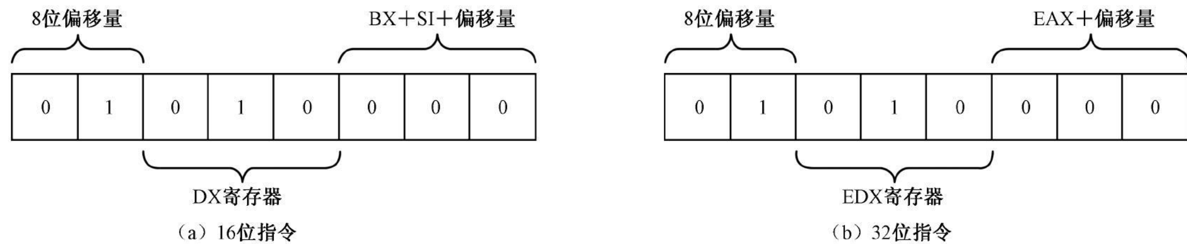


图10-8　16 位指令和32 位指令的寻址方式和操作数类型编码对比

在编写程序的时候，就应当考虑到指令的运行环境。为了指明程序的默认运行环境，编译器提供了伪指令bits，用于指明其后的指令应该被编译成16 位的，还是32 位的。比如：

```
bits 16
mov cx,dx      ;89 D1
mov eax,ebx    ;66 89 D8

bits 32
mov cx,dx      ;66 89 D1
mov eax,ebx    ;89 D8
```

注意，bits 16 或者bits 32 可以放在方括号中，也可以没有方括号。以下两种方式都是允许的：

```
[bits 32]
mov ecx,edx

bits 16
mov ax,bx
```

最后，16 位模式是默认的编译模式。如果没有指定指令的编译模式，则默认是“bits 16”的。

有关寻址方式和指令前缀的话题比较复杂，在后面的章节里，我们将在适当的时候，结合程序和具体的指令进行讲解。

10.3.3 一般指令的扩展

由于32 位的处理器都拥有32 位的寄存器和算术逻辑部件，而且同内存芯片之间的数据通路至少是32 位的，因此，所有以寄存器或者内存单元为操作数的指令都被扩充，以适应32 位的算术逻辑操作。而且，这些扩展的操作即使是在16 位模式下（实模式和16 位保护模式）也是可用的。比如加法指令ADD，在32 位处理器上，除了允许8 位或者16 位的操作数外，32 位的操作数现在也是可用的：

```
add al,bl
add ax,bx
```

```
add eax,ebx
add dword [ecx],0x0000005f
```

除了双操作数指令，单操作数指令也同样允许32 位操作数。比如：

```
inc al
inc dword [0x2000]
dec dword [eax*2]
```

我们已经接触过的逻辑移动指令，如shl、shr 等，目的操作数也扩展至32 位，但用于指定移动次数的源操作数足够应付32 位的环境，没有变化。举例：

```
shl eax,1
shl eax,9
shl dword [eax*2+0x08],cl
```

和16 位时代一样，在32 位处理器上，逻辑移动指令的源操作数如果是寄存器的话，则依然必须使用CL。同时，32 位处理器在实际执行时，要先将源操作数（在CL 寄存器内）同0x1F 做逻辑与。也就是说，仅保留源操作数的低5 位，因此，实际移动的次数最大为31。

在16 位处理器上，loop 指令的循环次数在寄存器CX 中。在32 位处理器上，如果当前的运行模式是16 位的（bits 16，8086 实模式或者16 位保护模式），那么，loop 指令执行时，依然使用CX 寄存器；否则，如果运行在32 位模式下（bits 32），则使用的是ECX 寄存器。

在16 位处理器上，无符号数乘法指令mul 的格式为

```
mul r/m8        ;AX ← AL×r/m8
mul r/m16       ;DX:AX ← AX×r/m16
```

在32 位处理器上，除了依然支持上述操作外，还支持以下扩展的格式：

```
mul r/m32       ;EDX:EAX ← EAX×r/m32
```

这样，两个32 位的数相乘，得到一个64 位的结果。这里有个例子：

```
mov eax,0x10000
mov ebx,0x20000
mul ebx
```

有符号数乘法指令imul 与此相同。

相应地，无符号数和有符号数除法也做了32 位扩展：

```
div r/m32
idiv r/m32
```

在这里，被除数是64 位的，高32 位在EDX 寄存器；低32 位在EAX 寄存器。除数是32 位的，位于32 位的寄存器，或者存放有32 位实际操作数的内存地址。指令执行后，32 位的商在EAX 寄存器，32 位的余数在EDX 寄存器。

32 位处理器的栈操作指令push 和pop 也有所扩展，允许压入双字操作数。特别是，它现在支持立即数压栈操作。立即数压栈操作的指令格式为

```
push imm8        ;操作码为 6A
push imm16       ;操作码为 68
push imm32       ;操作码为 68
```

举个例子可能更清楚一些。比如：

```
push byte 0x55
```

在这里，关键字“byte”仅仅是给编译器用的，告诉它，压入的是字节（毕竟立即数0x55 可以解释为字0x0055 或者双字0x00000055），而不是用来在编译后的机器指令前添加指令前缀。

这条指令的16 位形式（用bits 16 编译）和32 位形式（用bits 32 编译）是一样的，机器代码都是

```
6A 55
```

但是，当它执行时，就不同了。注意，无论在什么时候，处理器都不会真的压入一字节，要么压入字，要么压入双字。因此，在16 位模式下，默认的操作数字长是16，处理器在执行时，将该字节的符号位扩展到高8 位，然后压入栈，压栈时使用SP 寄存器，且先将SP 的内容减去2。这就是说，实际压入栈中的数值是0x0055；在32 位模式下，压入的

内容是该字节操作数符号位扩展到高24 位的结果，即0x00000055。压栈时使用ESP 寄存器，且先将ESP 的内容减去4。

如果压入的是字操作数，则必须用关键字“word”来修饰。如：

```
push word 0xfffb
```

在16 位模式下，默认的操作数字长是16，处理器在执行时，直接压入该字，压栈时使用SP 寄存器，且先将SP 的内容减去2；在32 位模式下，压入的内容是该操作数符号位扩展到高16 位的结果，即0xFFFFFFFB，压栈时使用ESP 寄存器，且先将ESP 的内容减去4。

如果压入的是双字操作数，则必须用关键字“dword”来修饰。如：

```
push dword 0xfb
```

则无论是在16 位模式下，还是在32 位模式下，压入的都是0x000000FB，而且栈指针寄存器（SP 或者ESP）都先减去4。

对于实际操作数位于通用寄存器，或者位于内存单元的情况，只能压入字或者双字，指令格式为：

```
push r/m16
push r/m32
```

如果是寄存器，则可以使用16 位或者32 位的通用寄存器。比如：

```
push ax
push edx
```

如果被压入的16 位或者32 位操作数位于内存单元中，则必须用关键字“word”或者“dword”修饰，以指示操作数的大小：

```
push word [0x2000]
push dword [ecx+esi*2+0x02]
```

无论被压入的数位于寄存器，还是位于内存单元，在16 位模式下，如果压入的是字操作数，那么先将SP 的内容减去2；如果压入的是双字，应当先将SP 的内容减去4。在32 位模式下，如果压入的是字操作

数，那么先将ESP 的内容减去2；如果压入的是双字，应当先将ESP 的内容减去4。

压入段寄存器的操作比较特殊。以下是压入段寄存器的push 指令格式：

```
push cs          ;机器指令为 0E
push ds          ;机器指令为 1E
push es          ;机器指令为 06
push fs          ;机器指令为 0F A0
push gs          ;机器指令为 0F A8
push ss          ;机器指令为 16
```

在16 位模式下，先将SP 的内容减去2，然后直接压入段寄存器的内容；在32 位模式下，要先将段寄存器的内容用零扩展到32 位，即高16 位为全零。然后，将ESP 的内容减去4，再压入扩展后的32 位值。

本 章 习 题

1．在编译阶段，如果指定的编译模式是bits 16，那么，mov bx,16的机器码为BB 10 00。相反，mov ebx,16 的机器码为66 BB 10 00 00 00。试问，如果指定了编译模式bits 32，这两条指令编译后的机器码又分别是什么？

2．以下程序片断：

```
bits 16
mov bx,16        ;BB 10 00
mul bx           ;F7 E3
```

将生成机器指令序列BB 10 00 F7 E3。

当处理器在32 位保护模式下执行这些代码时，会有什么问题？

第11章　进入保护模式

一般来说，操作系统负责整个计算机软、硬件的管理，它做任何事情都是可以的。但是，用户程序却应当有所限制，只允许它访问属于自己的数据，即使是转移，也只允许在自己的各个代码段之间进行。

问题在于，在实模式下，用户程序对内存的访问非常自由，没有任何限制，随随便便就可以修改任何一个内存单元。比如以下代码片断，它先保存当前的数据段地址，然后修改别人的数据，最后再回到原先的数据段：

```
mov cx,0x8000      ;逻辑段地址
push ds
mov ds,cx
mov [0x05],dx      ;逻辑地址 0x8000:0x0005，即物理地址 0x80005
pop ds
```

很显然，即使物理内存单元0x80005 不属于当前程序，它照样可以切换到那里，并随意修改其中的内容。最恐怖的是，如果那个地方是操作系统或其他用户程序的“地盘”，那将带来不可预料的后果。通过这个例子，你就知道为什么很多人能通过修改内存中的数据来提升游戏人物的法力和生命值，并获得各种道具。

在多用户、多任务时代，内存中会有多个用户（应用）程序在同时运行。为了使它们彼此隔离，防止因某个程序的编写错误或者崩溃而影响到操作系统和其他用户程序，使用保护模式是非常有必要的。

本章学习目标：

1．了解x86 处理器的保护模式需要先定义全局描述符表GDT，认识段描述符的各个组成部分以及它们的含义和作用。

2．认识32 位处理器的全局描述符表寄存器GDTR、段寄存器（由段选择器和描述符高速缓存器组成）、控制寄存器CR0 和段选择子。

3．了解进入32 位保护模式的方法和步骤。

4．学习保护模式下的一些程序调试技术，如察看全局描述符表GDT、段寄存器和控制寄存器等。

5．学习一条x86 处理器的新指令lgdt。

11.1　代码清单11-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：11-1（主引导扇区程序）

源程序文件：c11_mbr.asm

11.2　全局描述符表

我们知道，为了让程序在内存中能自由浮动而又不影响它的正常执行，处理器将内存划分成逻辑上的段，并在指令中使用段内偏移地址。在保护模式下，对内存的访问仍然使用段地址和偏移地址，但是，在每个段能够访问之前，必须先进行登记。

这种情况好有一比。就像是开公司做生意，在实模式下，开公司不需要登记，卖什么都没有人管，随时都可以开张。但在保护模式下就不行了，开公司之前必须先登记，登记的信息包括住址（段的起始地址）、经营项目（段的界限等各种访问属性）。这样，每当你做的买卖和你的注册项目不符时，就会被阻止。对段的访问也是一样，当你访问的偏移地址超出段的界限时，处理器就会阻止这种访问，并产生一个叫做内部异常的中断。

和一个段有关的信息需要8 个字节来描述，所以称为段描述符（Segment Descriptor），每个段都需要一个描述符。为了存放这些描述符，需要在内存中开辟出一段空间。在这段空间里，所有的描述符都是挨在一起，集中存放的，这就构成一个描述符表。

最主要的描述符表是全局描述符表（Global Descriptor Table，GDT），所谓全局，意味着该表是为整个软硬件系统服务的。在进入保护模式前，必须要定义全局描述符表。

如图11-1 所示，为了跟踪全局描述符表，处理器内部有一个48 位的寄存器，称为全局描述符表寄存器（GDTR）。该寄存器分为两部分，分别是32 位的线性地址和16 位的边界。32 位的处理器具有32 根地址线，可以访问的地址范围是0x00000000 到0xFFFFFFFF，共232 字节的内存，即4GB 内存。所以，GDTR 的32 位线性基地址部分保存的是全局描述符表在内存中的起始线性地址，16 位边界部分保存的是全局描述符表的边界（界限），其在数值上等于表的大小（总字节数）减一。

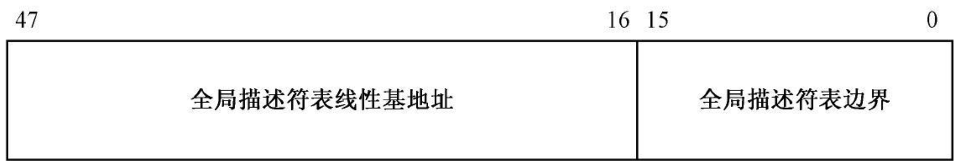


图11-1　全局描述符表寄存器GDTR

换句话说，全局描述符表的界限值就是表内最后1 字节的偏移量。第1 字节的偏移量是0，最后1 字节的偏移量是表大小减一。如果界限值为0，表示表的大小是1 字节。

因为GDT 的界限是16 位的，所以，该表最大是216 字节，也就是65536 字节（64KB）。又因为一个描述符占8 字节，故最多可以定义8192 个描述符。实际上，不一定非得这么多，到底有多少，视需要而定，但最多不能超过8192 个。

理论上，全局描述符表可以位于内存中的任何地方。但是，如图11-2 所示，由于在进入保护模式之后，处理器立即要按新的内存访问模式工作，所以，必须在进入保护模式之前定义GDT。但是，由于在实模式下只能访问1MB 的内存，故GDT 通常都定义在1MB 以下的内存范围中。当然，允许在进入保护模式之后换个位置重新定义GDT。

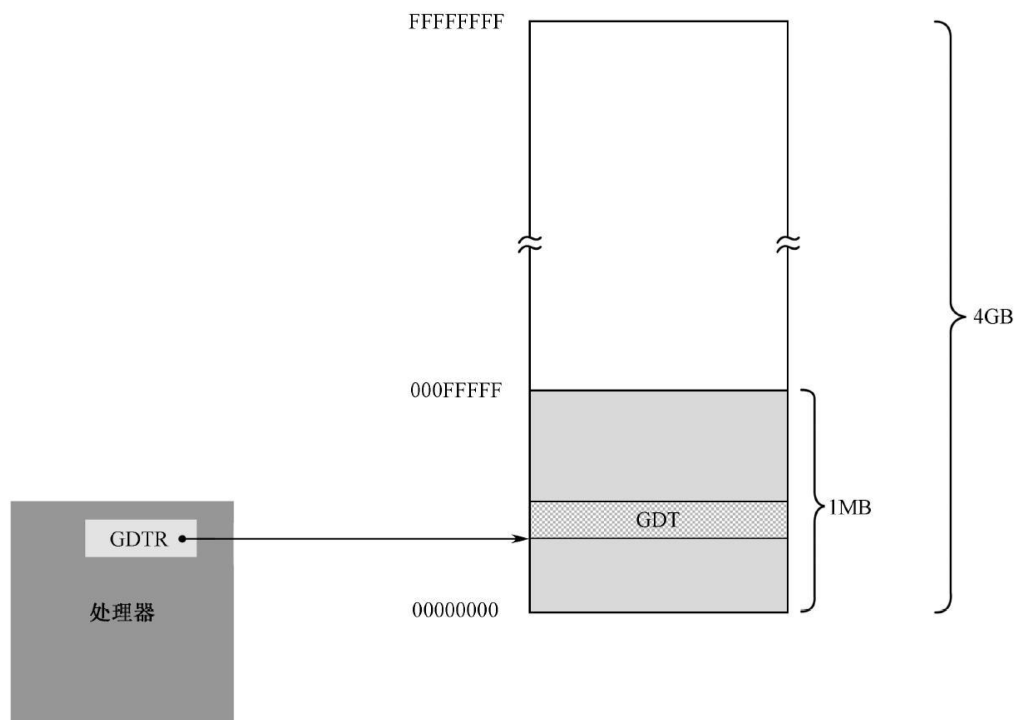


图11-2　GDT 和GDTR 的关系示意图

11.3　存储器的段描述符

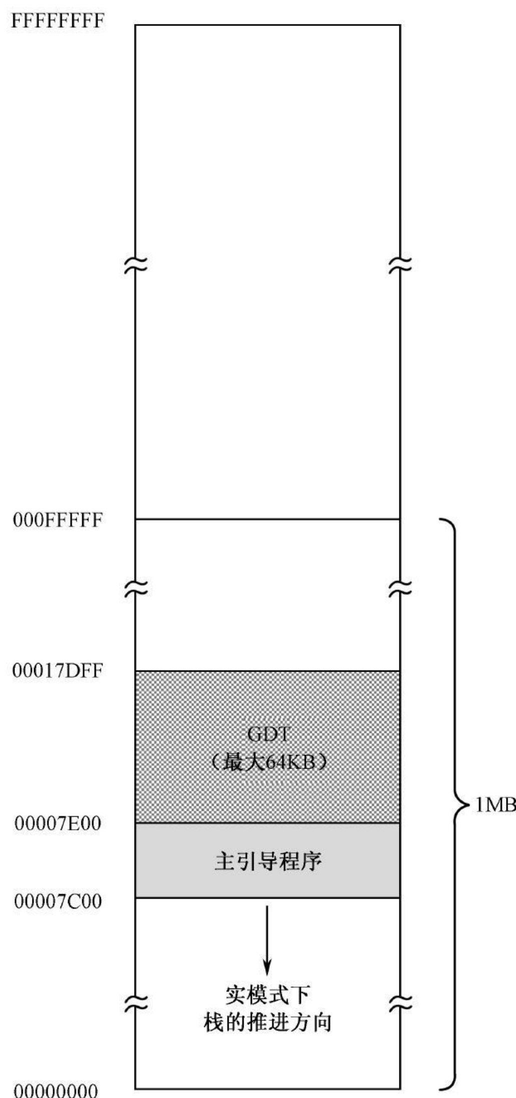


图11-3　进入保护模式前的内存映象

和往常一样，在程序的开始部分要初始化段寄存器。代码清单11-1 第7～9行用于初始化栈，使栈段的逻辑段地址和代码段相同，并使栈指针寄存器SP 指向0x7c00。这是个分界线，从这里，代码向上扩展，而栈向下扩展。

下面开始定义主引导扇区代码所使用的数据段、代码段和栈段。在保护模式下，内存的访问机制完全不同，即，必须通过描述符来进行。所以，这些段必须重新在GDT 中定义。

先是确定GDT 的起始线性地址。代码清单11-1 第96 行，声明了标号gdt_base 并初始化了一个双字0x00007e00，我们决定从这个地方开始创建全局描述符表（GDT）。这是有意的，如图11-3 所示，在实模式下，主引导程序的加载位置是0x0000:0x7c00，也就是物理地址0x07c00。因为现在的地址是32 位的，所以它现在对应着物理地址0x00007c00。主引导扇区程序共512（0x200）字节，所以，我们决定把GDT 设在主引导程序之后，也就是物理地址0x00007e00 处。因为GDT 最大可以为64KB，所以，理论上，它的尺寸可以扩展到物理地址0x00017dff 处。

相应地，因为栈指针寄存器SP 被初始化为0x7c00，和CS 一样，栈段寄存器SS 被初始化为0x0000，而且栈是向下扩展的，所以，从0x00007c00 往下的区域是实际上可用的栈区域。只不过，该区域包含了很多BIOS 数据，包括实模式下的中断向量表，所以一定要小心。这是没有办法的事，在实模式下，处理器不会为此负责，只能靠你自己。

实模式和保护模式在内存访问上是有区别的，在保护模式下，你不能说访问哪个段就访问哪个段，在访问之前，必须先在GDT 内定义要访问的内存段。也许你觉得多此一举，“想访问哪段内存，我就在GDT 中定义一个描述符，这和直接访问有什么区别？反正也能随心所欲，只不过多了一道手续，这又谈何限制和保护呢？”

实际上并非如此。如果整个计算机系统中只有一个程序在工作，那当然是正确的。问题在于，会有很多程序共同在操作系统上运行。想想你平时玩的电子游戏、音视频播放器、WPS、Word、Excel，它们都依靠Windows 的支撑才能运行。所以，描述符不是由用户程序自己建立的，而是在加载时，由操作系统根据你的程序结构而建立的，而用户程序通常是无法建立和修改GDT 的，也就只能老老实实地在自己的地盘上工作。在这种情况下，操作系统为你的程序建立了几个段，你就只能在这些段内工作，超出这个范围，或者未按预定的方法访问这些段，都将被处理器阻止。

一旦确定了GDT 在内存中的起始位置，下一步的工作就是确定要访问的段，并在GDT 中为这些段创建各自的描述符。

如图11-4 所示，每个描述符在GDT 中占8 字节，也就是2 个双字，或者说是64 位。图中，下面是低32 位，上面是高32 位。



图11-4　存储器的段描述符格式

很明显，描述符中指定了32 位的段起始地址，以及20 位的段边界。在实模式下，段地址并非真实的物理地址，在计算物理地址时，还要左移4 位（乘以16）。和实模式不同，在32 位保护模式下，段地址是32 位的线性地址，如果未开启分页功能，该线性地址就是物理地址。页功能将在第16 章和第17 章讲解，而且开启页功能需要做很多准备工作。目前，如果没有特别说明，线性地址就是物理地址。描述符中的段基地址和段界限不是连续的，把它们分成几段似乎不科学。但这也是没有办法

的事，这是从80286 处理器上带来的后遗症。80286 也是16 位的处理器，也有保护模式，但属于16 位的保护模式。而且，其地址是24 位的，允许访问最多16MB 的内存。尽管80286 的16 位保护模式从来也没形成气候，但是，32 位处理器为了保持同80286 的兼容，只能在旧描述符的格式上进行扩充，这是不得已的做法。

段基地址可以是0～4GB 范围内的任意地址，不过，还是建议应当选取那些16 字节对齐的地址。尽管对于Intel 处理器来说，允许不对齐的地址，但是，对齐能够使程序在访问代码和数据时的性能最大化。这一点，对于那些学过计算机原理，特别是了解内存芯片组织的人来说，是最清楚不过的。

20 位的段界限用来限制段的扩展范围。因为访问内存的方法是用段基地址加上偏移量，所以，对于向上扩展的段，如代码段和数据段来说，偏移量是从0 开始递增，段界限决定了偏移量的最大值；对于向下扩展的段，如栈段来说，段界限决定了偏移量的最小值。

G 位是粒度（Granularity）位，用于解释段界限的含义。当G 位是“0”时，段界限以字节为单位。此时，段的扩展范围是从1 字节到1 兆字节（1B～1MB），因为描述符中的界限值是20 位的。相反，如果该位是“1”，那么，段界限是以4KB 为单位的。这样，段的扩展范围是从4KB 到4GB。

S 位用于指定描述符的类型（Descriptor Type）。当该位是“0”时，表示是一个系统段；为“1”时，表示是一个代码段或者数据段（栈段也是特殊的数据段）。系统段将在以后介绍。

DPL 表示描述符的特权级（Descriptor Privilege Level，DPL）。这两位用于指定段的特权级。共有4 种处理器支持的特权级别，分别是0、1、2、3，其中0 是最高特权级别，3 是最低特权级别。刚进入保护模式时执行的代码具有最高特权级0（可以看成是从处理器那里继承来的），这些代码通常都是操作系统代码，因此它的特权级别最高。每当操作系统加载一个用户程序时，它通常都会指定一个稍低的特权级，比如3 特权级。不同特权级别的程序是互相隔离的，其互访是严格限制的，而且有些处理器指令（特权指令）只能由0 特权级的程序来执行，为的就是安全。

在这里，描述符的特权级用于指定要访问该段所必须具有的最低特权级。如果这里的数值是2，那么，只有特权级别为0、1 和2 的程序才能

访问该段，而特权级为3 的程序访问该段时，处理器会予以阻止。特权级将在以后专门讲解，谁也不希望自己的特权级最低，何况现在有随便决定段特权级别的自由。那么，好吧，我们现在一律将特权级设定为最高的0。

P 是段存在位（Segment Present）。P 位用于指示描述符所对应的段是否存在。一般来说，描述符所指示的段都位于内存中。但是，当内存空间紧张时，有可能只是建立了描述符，对应的内存空间并不存在，这时，就应当把描述符的P 位清零，表示段并不存在。另外，同样是在内存空间紧张的情况下，会把很少用到的段换出到硬盘中，腾出空间给当前急需内存的程序使用（当前正在执行的），这时，同样要把段描述符的P 位清零。当再次轮到它执行时，再装入内存，然后将P位置1。

P 位是由处理器负责检查的。每当通过描述符访问内存中的段时，如果P 位是“0”，处理器就会产生一个异常中断。通常，该中断处理过程是由操作系统提供的，该处理过程的任务是负责将该段从硬盘换回内存，并将P 位置1。在多用户、多任务的系统中，这是一种常用的虚拟内存调度策略。当内存很小，运行的程序很多时，如果计算机的运行速度变慢，并伴随着繁忙的硬盘操作时，说明这种情况正在发生。

D/B 位是“默认的操作数大小”（Default Operation Size）或者“默认的栈指针大小”（Default Stack Pointer Size），又或者“上部边界”（Upper Bound）标志。

设立该标志位，主要是为了能够在32 位处理器上兼容运行16 位保护模式的程序。尽管这种程序现在已经非常罕见了，但它毕竟存在过，兼容，这是Intel 公司能够兴旺发达的重要因素。

该标志位对不同的段有不同的效果。对于代码段，此位称做“D”位，用于指示指令中默认的偏移地址和操作数尺寸。D＝0 表示指令中的偏移地址或者操作数是16 位的；D＝1，指示32 位的偏移地址或者操作数。

举个例子来说，如果代码段描述符的D 位是0，那么，当处理器在这个段上执行时，将使用16 位的指令指针寄存器IP 来取指令，否则使用32 位的EIP。

对于栈段来说，该位被叫做“B”位，用于在进行隐式的栈操作时，是使用SP 寄存器还是ESP 寄存器。隐式的栈操作指令包括push、pop 和call 等。如果该位是“0”，在访问那个段时，使用SP 寄存器，否则就是使用ESP 寄存器。同时，B 位的值也决定了栈的上部边界。如果B＝0，那

么栈段的上部边界（也就是SP 寄存器的最大值）为0xFFFF；如果B＝1，那么栈段的上部边界（也就是ESP 寄存器的最大值）为0xFFFFFFFF。

对于本书来说，它应当为“1”。本书不过多涉及16 位保护模式，它已经非常罕见了。

L 位是64 位代码段标志（64-bit Code Segment），保留此位给64 位处理器使用。目前，我们将此位置“0”即可。

TYPE 字段共4 位，用于指示描述符的子类型，或者说是类别。如表11-1 所示，对于数据段来说，这4 位分别是X、E、W、A 位；而对于代码段来说，这4 位则分别是X、C、R、A 位。

表11-1　代码段和数据段描述符的TYPE 字段

X	E	W	A	描述符类别	含　义
0	0	0	×	数据	只读
0	0	1	×		读、写
0	1	0	×		只读，向下扩展
0	1	1	×		读、写，向下扩展
X	C	R	A	描述符类别	含　义
1	0	0	×	代码	只执行
1	0	1	×		执行、读
1	1	0	×		只执行，依从的代码段
1	1	1	×		执行、读，依从的代码段

表11-1 中，X 表示是否可以执行（eXecutable）。数据段总是不可执行的，X＝0；代码段总是可以执行的，因此，X＝1。

对于数据段来说，E 位指示段的扩展方向。E＝0 是向上扩展的，也就是向高地址方向扩展的，是普通的数据段；E＝1 是向下扩展的，也就是向低地址方向扩展的，通常是栈段。W 位指示段的读写属性，或者说段是否可写，W＝0 的段是不允许写入的，否则会引发处理器异常中断；W ＝1 的段是可以正常写入的。

对于代码段来说，C 位指示段是否为特权级依从的（Conforming）。C＝0 表示非依从的代码段，这样的代码段可以从与它特权级相同的代码段调用，或者通过门调用；C＝1 表示允许从低特权级的程序转移到该段执行。关于特权级和特权级检查的知识将在第14 章介绍。R 位指示代码段是否允许读出。代码段总是可以执行的，但是，为了防止程序被破坏，它是不能写入的。至于是否有读出的可能，由R 位

指定。R＝0 表示不能读出，如果企图去读一个R＝0 的代码段，会引发处理器异常中断；如果R＝1，则代码段是可以读出的，即可以把这个段的内容当成ROM 一样使用。

也许有人会问，既然代码段是不可读的，那处理器怎么从里面取指令执行呢？事实上，这里的R 属性并非用来限制处理器，而是用来限制程序和指令的行为。一个典型的例子是使用段超越前缀“CS:”来访问代码段中的内容。

数据段和代码段的A 位是已访问（Accessed）位，用于指示它所指向的段最近是否被访问过。在描述符创建的时候，应该清零。之后，每当该段被访问时，处理器自动将该位置“1”。对该位的清零是由软件（操作系统）负责的，通过定期监视该位的状态，就可以统计出该段的使用频率。当内存空间紧张时，可以把不经常使用的段退避到硬盘上，从而实现虚拟内存管理。

AVL 是软件可以使用的位（Available），通常由操作系统来用，处理器并不使用它。如果你把它理解成“好吧，该安排的都安排了，最后多出这么一位，不知道干什么用好，就给软件用吧”，我也不反对，也许Intel 公司也不会说些什么。

11.4　安装存储器的段描述符并加载GDTR

现在开始安装各个描述符，让我们回到代码清单11-1。

不要忘了，我们现在还处于实模式下。因此，在GDT 中安装描述符，必须将GDT 的线性地址（物理地址）转换成逻辑段地址和偏移地址。

GDT 的线性地址是我们直接给出的，放在程序中的标号gdt_base处。第12 行，将GDT 线性基地址的低16 位传送到寄存器AX 中。和从前一样，这里使用了段超越前缀“cs：”，表明是访问代码段中的数据；又因为主引导程序的实际加载位置是逻辑地址0x0000:0x7c00，故标号gdt_base处的偏移地址是gdt_base+0x7c00。

同样地，第13 行将GDT 线性基地址的高16 位传送到寄存器DX。

第14～17 行将线性基地址转换成逻辑地址，方法是将DX:AX 除以16，得到的商是逻辑段地址，余数是偏移地址。接着，将AX 中的逻辑段地址传送到数据段寄存器DS 中，将偏移地址传送到寄存器BX 中。

处理器规定，GDT 中的第一个描述符必须是空描述符，或者叫哑描述符或NULL 描述符，相信后者对于有C 语言经历的读者来说更容易接受。

很多时候，寄存器和内存单元的初始值会为0，再加上程序设计有问题，就会在无意中用全0的索引来选择描述符。因此，处理器要求将第一个描述符定义成空描述符。

为此，第20、21 行将两个全0 的双字分别写入偏移地址为BX 和BX+4 的地方。

进入保护模式之后必然要从一个代码段开始执行。现在就来定义代码段描述符。

第24、25 行，接着安装代码段描述符，该描述符的低32 位是0x7c0001ff，高32 位是0x00409800。结合图11-4 可以分析出，该段的基本情况为：

```
线性基地址为 0x00007C00。
段界限为 0x001FF，粒度为字节（G＝0）。该段的长度为 512 字节。
属于存储器的段（S＝1）。
这是一个 32 位的段（D＝1）。
该段目前位于内存中（P＝1）。
段的特权级为 0（DPL＝00）。
这是一个只能执行的代码段（TYPE=1000）。
```

很明显，该描述符所指向的段，就是现在正在执行的主引导程序所在的区域。如图11-5 所示，这是描述符各字节在内存中的映象。Intel 处理器是低端字节序的，所以低双字在低地址端，高双字在高地址端；低字在低地址端，高字在高地址端；低字节在低地址端，高字节在高地址端。

第28、29 行，用于安装一个数据段的描述符。对照图11-4，很明显，这个段具有以下性质：

```
线性基地址为 0x000B8000。
段界限为 0x0FFFF，粒度为字节（G＝0）。即，该段的长度为 64KB。
属于存储器的段（S＝1）。
这是一个 32 位的段（D＝1）。
该段目前位于内存中（P＝1）。
段的特权级为 0（DPL＝00）。
```

```
这是一个可读可写、向上扩展的数据段（TYPE=0010）。
```

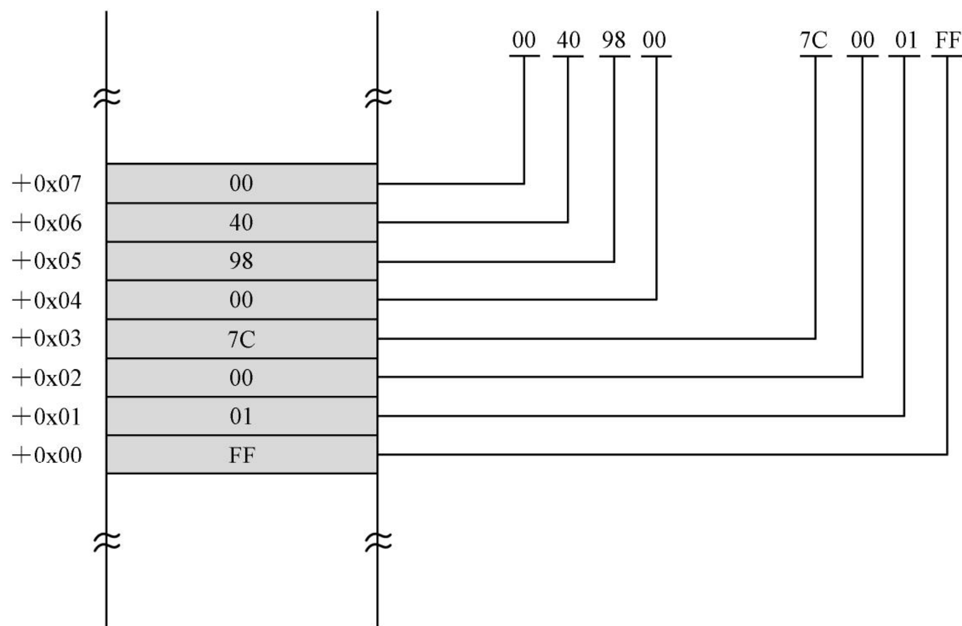


图11-5　描述符各个字节在内存中的映象

我们用程序在屏幕上显示内容已经不是一次两次了，很容易看出，线性地址0x000b8000 就是显存的起始地址，看起来，我们要用这个段来显示字符。

第32、33 行，用于安装栈段的描述符。对照图11-4，该段的性质如下：

```
线性基地址为 0x00000000。
段界限为 0x07A00，粒度为字节（G=0）。
属于存储器的段（S=1）。
这是一个 32 位的段（D=1）。
该段目前位于内存中（P=1）。
段的特权级为 0（DPL=00）。
这是一个可读可写、向下扩展的数据段，在这里是栈段（TYPE=0010）。
```

在这里，段界限的值0x07a00 加上1（0x07a01），就是ESP 寄存器所允许的最小值。当执行push、call 这样的隐式栈操作时，处理器会检查ESP 寄存器的值，一旦发现它小于等于这里指定的数值，会引发异常中断。关于栈界限的讨论将在本章的后面接着进行。

好了，现在所有的描述符都已经安装完毕，接下来的工作是加载描述符表的线性基地址和界限到GDTR 寄存器，这要使用lgdt 指令，该指令的格式为

```
        lgdt m48        ;lgdt m16&m32
```

这就是说，该指令的操作数是一个48 位（6字节）的内存区域。在16 位模式下，该地址是16 位的；在32 位模式下，该地址是32 位的。该指令在实模式和保护模式下都可以执行。

在这6 字节的内存区域中，要求前（低）16 位是GDT 的界限值，后（高）32 位是GDT 的基地址。在初始状态下（计算机启动之后），GDTR 的基地址被初始化为0x00000000；界限值为0xFFFF。

该指向不影响任何标志位。

为此，代码清单11-1 第36 行，将GDT 表的界限值31 写入标号gdt_size 所在的内存单元。这里共有4 个描述符（包括空描述符），每个描述符占8 字节，一共是32 字节。GDT 表的界限值是表的总字节数减去一，所以是31。

接着，第38 行，把从标号gdt_size 开始的6 字节加载到GDTR 寄存器：

```
        lgdt [cs: gdt_size+0x7c00]
```

因为gdt_size 和gdt_base 是连续声明的，紧挨在一起，所以，从gdt_size 处读取6 个字节，就包括了gdt_base。注意，到目前为止，我们依然工作在实模式下，而且不要忘了，指令中的偏移地址都要加上0x7c00。可以在Bochs 中察看全局描述符表GDT 的内容，具体方法参见本章11.9.5 节。

检测点11.1

1．某描述符是64 位的0x004F9AFFFFFFFFFF，请问，段基地址是多少？段界限是多少？G、D、L、AVL、P、DPL、S 和TYPE 各是什么？

2．32 位保护模式下，某段为数据段，基地址为0x002FC0F0，段的长度为2MB，粒度为4KB，已经位于物理内存中，请给出其描述符的低32 位和高32 位。

11.5　关于第21 条地址线A20 的问题

在即将进入保护模式之前，这里还涉及一个历史遗留问题，那就是处理器的第21 根地址线，编号A20。“A”是Address 的首字符，就是地址，A0 是第一根地址线，A31 是第32 根地址线，所以，A20 就是第21 根地址线。在8086 处理器上运行程序不存在A20 问题，因为它只有20 根地址线。

实模式下的程序只能寻址1MB 内存，那是因为它依赖16 位的段地址左移4 位，加上16 位的偏移地址来访问内存。当逻辑段地址达到最大值0xFFFF 时，再加一，就会因进位而绕回到0x0000，因为段寄存器只能保留16 位的结果。至于段内偏移地址，也是如此。

这个问题可以从另一个角度来解释得更清楚一点。无论如何，从8086 处理器外部来看，每次当物理地址达到最高端0xFFFFF 时，再加一，结果为0x100000。但因为它只能维持20 位的地址，故进位自然丢失，地址又绕回最低地址端0x00000。程序员，你是知道的，他们喜欢钻研，更喜欢利用硬件的某些特性来展示自己的技术，很难说在当年有多少程序在依赖这个回绕特性工作着。

到了80286 时代，处理器有24 条地址线，地址回绕好像不灵了，因为比0x0FFFFF 大的数是0x100000，80286 处理器可以维持24 位的地址数据，进位不会被丢弃。那个时代，正是商业机器公司IBM 生意火红的时候，主导着个人计算机市场。为了能在80286 处理器上运行8086 程序而不会因地址线而产生问题，它们决定在主板上动一动手脚。

其实问题的解决办法很简单，只需要强制第21 根地址线恒为“0”就可以了。这样，0x0FFFFF 加1 的进位被强制为“0”，结果是0x000000；再加1，是0x000001，……，永远和实模式一样。

于是，如图11-6 所示，IBM 公司使用一个与门来控制第21 根地址线A20，并把这个与门的控制阀门放在键盘控制器内，端口号是0x60。向该端口写入数据时，如果第1 位是“1”，那么，键盘控制器通向与门的输出就为“1”，与门的输出就取决于处理器A20 是“0”还是“1”。

不过，这种做法非常烦琐，因为要访问键盘控制器，需要先判断状态，要等待键盘控制器不忙，至少需要十几个步骤，需要的指令数量比

本章的代码清单11-1 还多。

这种做法持续了若干年，直到80486 处理器推出后，才有了更快速的办法。相信在此期间，Intel 公司和IBM 公司都听到了不少的抱怨，为什么进入保护模式这么麻烦，一定要改改。从80486 处理器开始，处理器本身就有了A20M#引脚，意思是A20 屏蔽（A20 Mask），它是低电平有效的。

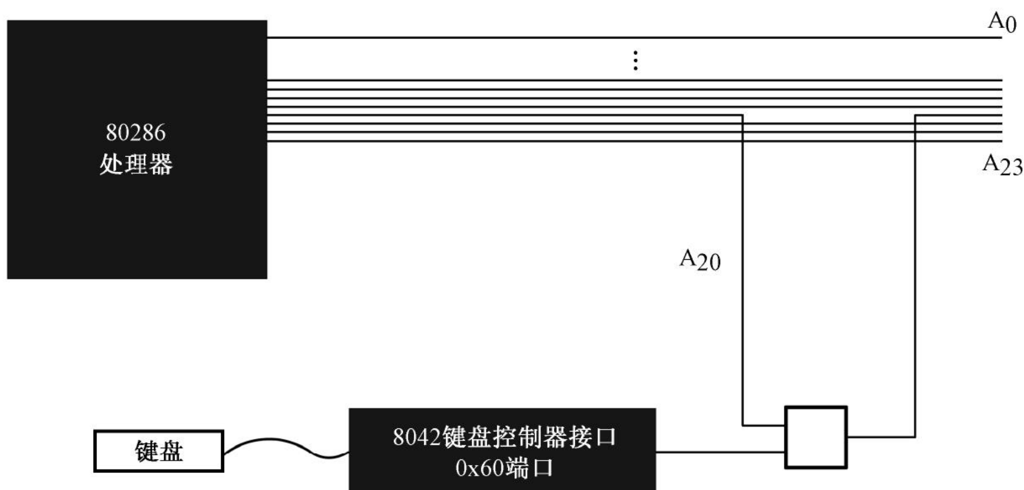


图11-6　早期的A20 控制策略

如图11-7 所示，输入输出控制器集中芯片ICH 的处理器接口部分，有一个用于兼容老式设备的端口0x92，第7～2 位保留未用，第0 位叫做INIT_NOW，意思是“现在初始化”，用于初始化处理器，当它从0 过渡到1 时，ICH 芯片会使处理器INIT#引脚的电平变低（有效），并保持至少16 个PCI 时钟周期。通俗地说，向这个端口写1，将会使处理器复位，导致计算机重新启动。

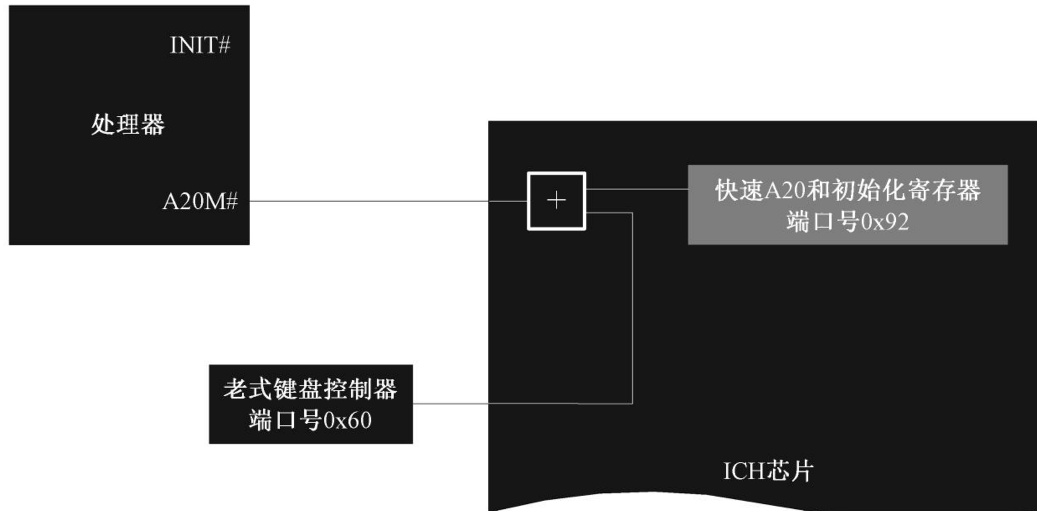


图11-7　改进后的A20 控制策略

端口0x92 的位1 用于控制A20，叫做替代的A20 门控制（Alternate A20 Gate，ALT_A20_GATE），它和来自键盘控制器的A20 控制线一起，通过或门连接到处理器的A20M#引脚。和使用键盘控制器的端口不同，通过0x92 端口显得非常迅速，也非常方便快捷，因此称为Fast A20。

当INIT_NOW 从0 过渡到1 时，ALT_A20_GATE 将被置“1”。这就是说，计算机启动时，第21 根地址线是自动启用的。A20M#信号仅用于单处理器系统，多核处理器一般不用。特别是考虑到传统的键盘控制器正逐渐被USB 键盘代替，这些老式设备也许很快就会消失。

接着来看代码清单11-1。

端口0x92 是可读写的，第40～42 行，先从该端口读出原数据，接着，将第2 位（位1）置“1”，然后再写入该端口，这样就打开了A20。

11.6 保护模式下的内存访问

一路披荆斩棘之后，你已经到达实模式和保护模式的分界线了。同时，你也会发现，控制这两种模式切换的开关原是在一个叫CR0 的寄存器。

CR0 是处理器内部的控制寄存器（Control Register，CR）。之所以有个“0”后缀，是因为还有CR1、CR2、CR3 和CR4 控制寄存器，甚至还有CR8。

CR0 是32 位的寄存器，包含了一系列用于控制处理器操作模式和运行状态的标志位。如图11-8 所示，它的第1 位（位0）是保护模式允许位（Protection Enable，PE），是开启保护模式大门的门把手，如果把该位置“1”，则处理器进入保护模式，按保护模式的规则开始运行。你可能会问，为什么只标识了一个PE 位，还把图画那么大。很简单，随着讲解的深入，我们还要接触其他标志位，把图的比例画得一致更好一些。

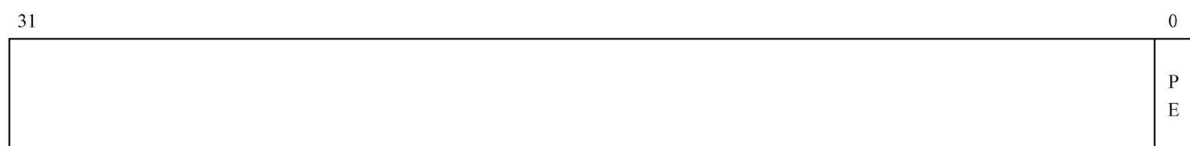


图11-8　控制寄存器CR0 的PE 位

保护模式下的中断机制和实模式不同，因此，原有的中断向量表不再适用，而且，必须要知道的是，在保护模式下，BIOS 中断都不能再用，因为它们是实模式下的代码。在重新设置保护模式下的中断环境之前，必须关中断，这就是第44 行的用意。

第46 行，将CRO 寄存器中的原有内容传送到寄存器EAX，准备修改它；第47 行，将它的第1 位（位0）置“1”，其他各位保持原来的状态不变；第48 行，将修改之后的内容重新写回CR0，这直接导致处理器的运行变成保护模式。

可以在Bochs 调试窗口中察看各个控制寄存器的内容，具体方法参见本章11.9.6 节。你可以在mov cr0,eax 指令执行前和执行后各察看一次，重点关注CR0 寄存器PE 位的前后变化。

我们知道，在实模式下，处理器访问内存的方式是将段寄存器的内容左移4 位，再加上偏移地址，以形成20 位的物理地址。

8086 处理器的段寄存器是16 位的，共有4 个：CS、DS、ES 和SS。而在32 位处理器内，在原先的基础上又增加了两个段寄存器FS 和GS。

如图11-9 所示，32 位处理器的这6 个段寄存器又分为两部分，前16 位和8086 相同，在实模式下，它们用于按传统的方式寻址1MB 内存，使用方法也没有变化，所以使得8086 的程序可以继续在32 位处理器上运行。同时，每个段寄存器还包括一个不可见的部分，称为描述符高速缓存器，用来存放段的线性基地址、段界限和段属性。既然不可见，那就是处理器不希望我们访问它。事实上，我们也没有任何办法来访问这些不可见的部分，它是由处理器内部使用的。

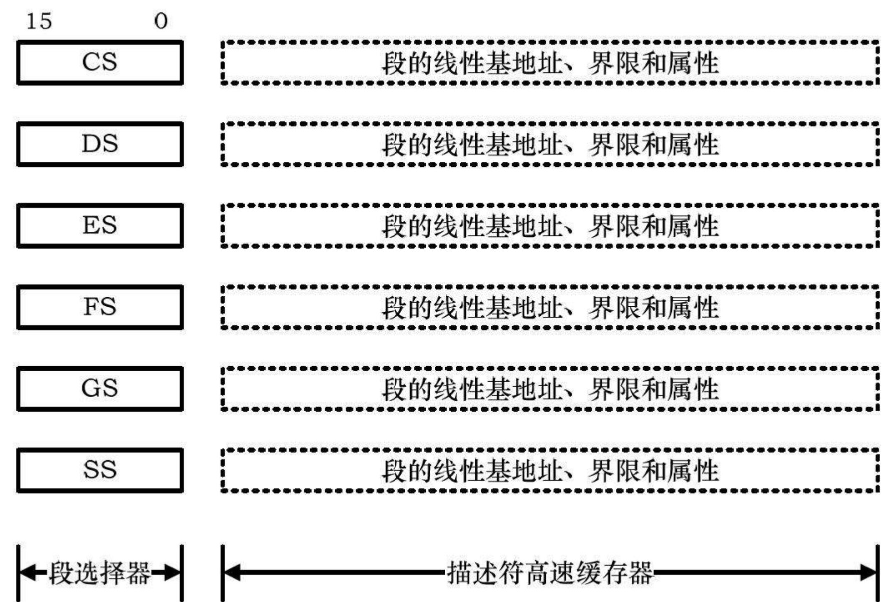


图11-9　32 位处理器内的段寄存器

在实模式下，访问内存用的是逻辑地址，即将段地址乘以16，再加上偏移地址。下面是一个例子：

```
mov cx,0x2000
mov ds,cx
mov [0xc0],al
mov cx,0xb800
mov ds,cx
mov [0x02],ah
```

以上，首先将段寄存器DS 的内容置为0x2000，这是逻辑段地址。接着，向该段内偏移地址为0x00c0 的地方写入1 字节（在寄存器AL 中），写入时，处理器将DS 的内容左移4 位，加上偏移地址，实际写入的物理地址是0x200c0。

在8086 处理器上，这是正确的。但是，在32 位处理器上，这个过程稍有不同。首先，每当引用一个段时，处理器自动将段地址左移4 位，并传送到描述符高速缓存器。此后，就一直使用描述符高速缓存器的内容作为段地址。所谓引用一个段，就是执行将段地址传送到段寄存器的指令。如

```
jmp 0xf000:0x5000
```

以上是引用代码段的一个例子，因为代码段的修改通常是用转移和调用指令进行的。如果是引用数据段，则一般采用以下形式：

```
mov ax,0x2000
mov ds,ax
```

只要不改变段寄存器DS 的内容，以后每次内存访问都直接使用DS 描述符高速缓存器中的内容。但是，在实模式下只能向段寄存器传送16 位的逻辑段地址（即，处理器不把它看成是描述符选择子），故，处理器仍然只能访问1MB 内存。也就是说，在实模式下，段寄存器描述符高速缓存器的内容仅低20 位有效，高12 位全部是零。

实模式下的6 个段寄存器CS、DS、ES、FS、GS 和SS，在保护模式下叫做段选择器。和实模式不同，保护模式的内存访问有它自己的方式。在保护模式下，尽管访问内存时也需要指定一个段，但传送到段选择器的内容不是逻辑段地址，而是段描述符在描述符表中的索引号。

如图11-10 所示，在保护模式下访问一个段时，传送到段选择器的是段选择子。它由三部分组成，第一部分是描述符的索引号，用来在描述

符表中选择一个段描述符。TI 是描述符表指示器（Table Indicator），TI＝0 时，表示描述符在GDT 中；TI＝1 时，描述符在LDT 中。LDT 的知识将在后面进行介绍，它也是一个描述符表，和GDT 类似。RPL 是请求特权级，表示给出当前选择子的那个程序的特权级别，正是该程序要求访问这个内存段。每个程序都有特权级别，也将在后面慢慢介绍，现在只需要将这两位置成“00”即可。

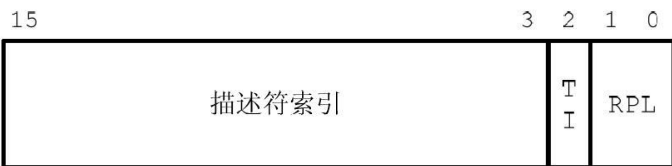


图11-10　段选择子的组成

为了说明保护模式下的内存访问，让我们回到代码清单11-1。前面已经创建了全局描述符表（GDT），而且在表中定义了4 个段描述符。数据段描述符在GDT 中的顺序是第3 个，因为编号都是从0 开始的，所以它的索引号（或者叫编号、槽位号）是2。

代码清单11-1 第56、57 行，将描述符选择子0x0010（二进制数0000_0000_ 00010_0_00）传送到段选择器DS 中。从选择子的二进制形式可以看出，指定的描述符索引号是2，指定的描述符表是GDT，请求特权级RPL 是00。

GDT 的线性基地址在GDTR 中，又因为每个描述符占8 字节，因此，描述符在表内的偏移地址是索引号乘以8。如图11-11 所示，当处理器在执行任何改变段选择器的指令时（比如pop、mov、jmp far、call far、iret、retf），就将指令中提供的索引号乘以8 作为偏移地址，同GDTR 中提供的线性基地址相加，以访问GDT。如果没有发现什么问题（比如超出了GDT 的界限），就自动将找到的描述符加载到不可见的描述符高速缓存部分。

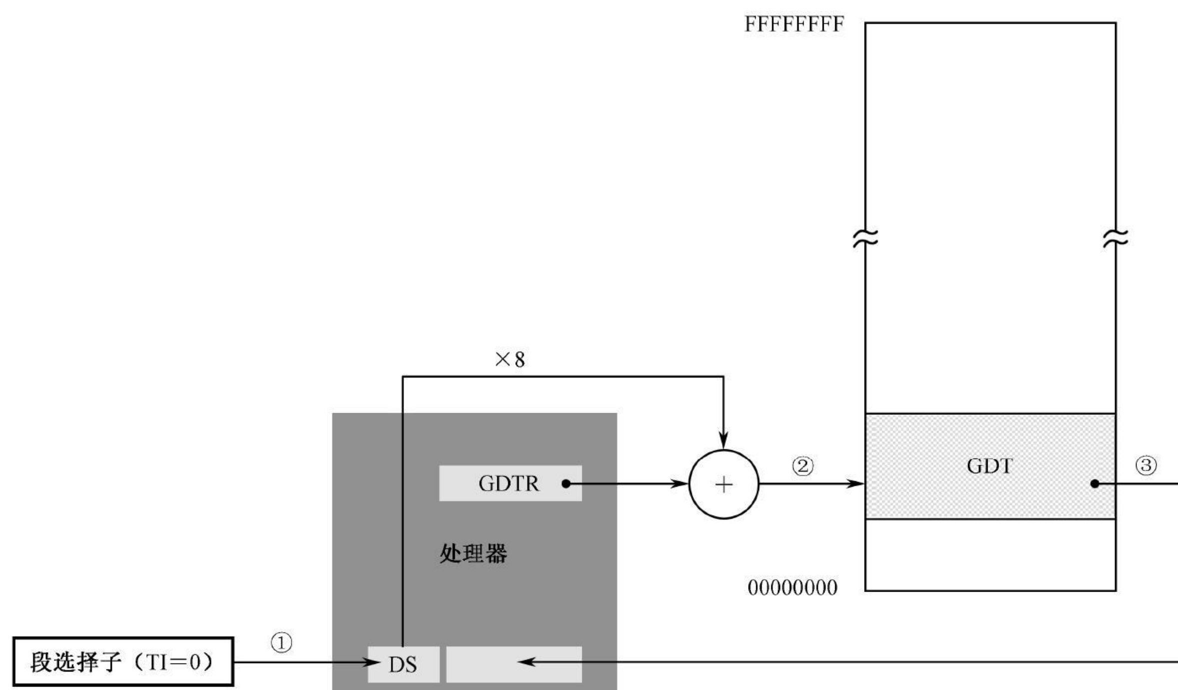


图11-11　段选择器和描述符高速缓存器的加载过程

加载的部分包括段的线性基地址、段界限和段的访问属性。在当前的例子中，线性基地址是0x000b8000，段界限是0x0ffff，段的属性是向上扩展，可读写的数据段，粒度为字节。

此后，每当有访问内存的指令时，就不再访问GDT 中的描述符，直接用当前段寄存器描述符高速缓存器提供线性基地址。因此，第60 行，因为指令中没有段超越前缀，故默认使用数据段寄存器DS。如图11-12 所示，执行这条指令时，处理器用DS 描述符高速缓存中的线性基地址加上指令中给出的偏移量0x00，形成32 位物理地址0x000b8000，并将字符“P”的ASCII 码写入该处。

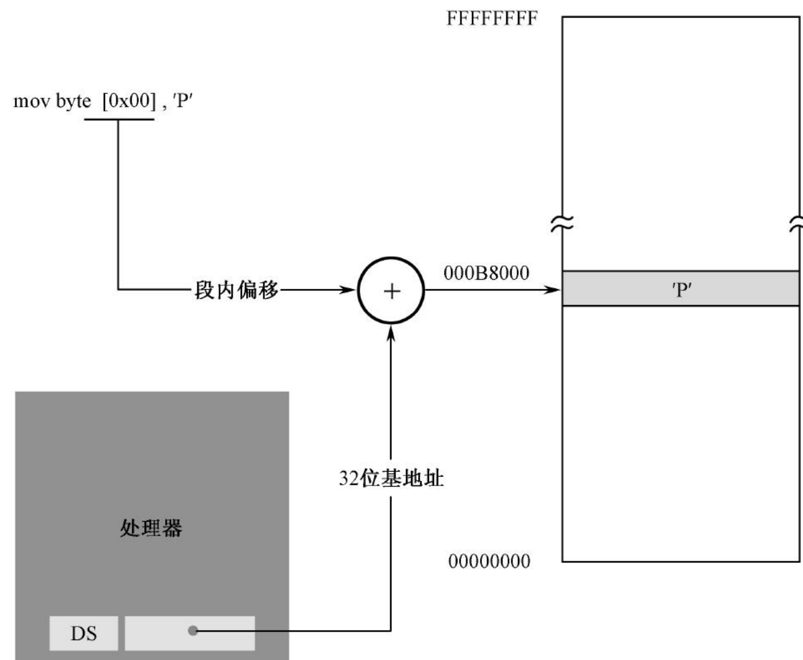


图11-12　保护模式下的内存访问示意图

不单单是访问数据段，即使是处理器取指令执行时，也采用了相同的方法。如图11-13 所示，在32 位保护模式下，处理器使用的指令指针寄存器是EIP。假设已经从描述符表中选择了一个段描述符，CS 描述符高速缓存器已经装载了正确的32 位线性基地址，那么，当处理器取指令时，会自动用描述符高速缓存器中的32 位线性基地址加上指令指针寄存器EIP 中的32 位偏移量，形成32 位物理地址，从内存中取得执令并加以执行。同时，EIP 的内容自动增加以指向下一条指令。当前指令执行完毕之后，处理器接着按上述方式取下一条指令加以执行。

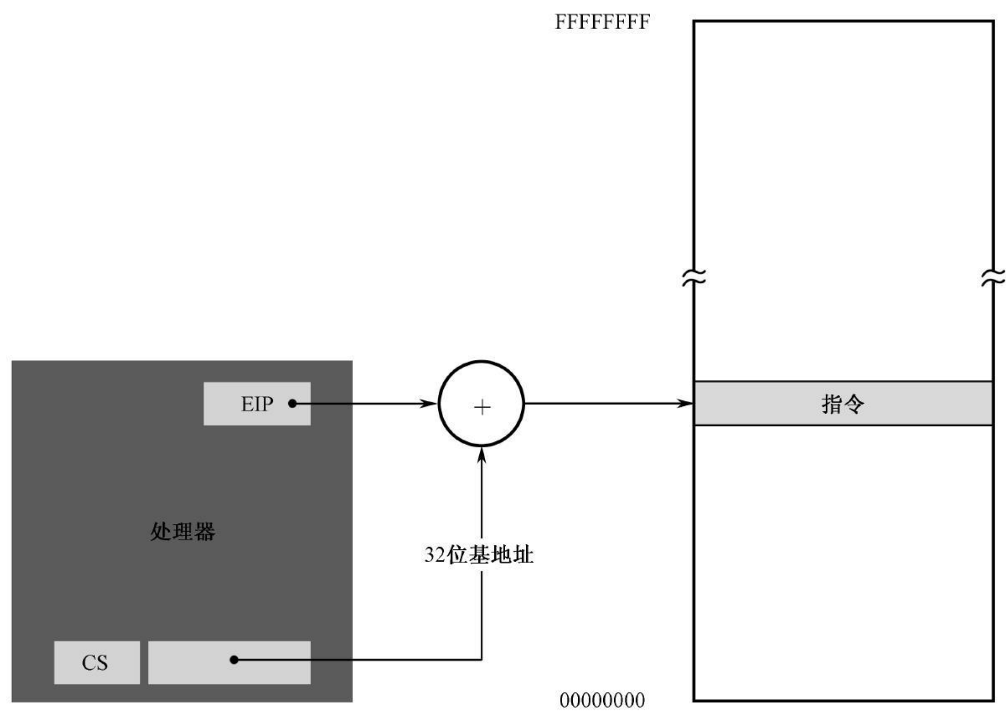


图11-13　保护模式下处理器取指令的过程示意图

11.7　清空流水线并串行化处理器

看起来我们所讲的内容有些超前了，毕竟前面刚刚设置了控制寄存器CR0 的PE 位，处理器刚刚切换到保护模式下。看起来一切都很简单，拧一下钥匙，汽车就发动了，不是吗？

不是这样的。这里有两个亟待解决的问题。

第一，正如上一节所述，即使是在实模式下，段寄存器的描述符高速缓存器也被用于访问内存，仅低20 位有效，高12 位是全零。当处理器进入保护模式后，这些内容依然残留着，但不影响使用，程序可以继续执行。但是，这些残留的内容在保护模式下是无效的，迟早会在执行某些指令的时候出问题。因此，比较安全的做法是尽快刷新CS、SS、DS、ES、FS 和GS 的内容，包括它们的段选择器和描述符高速缓存器。

第二，在进入保护模式前，有很多指令已经进入了流水线。因为处理器工作在实模式下，所以它们都是按16 位操作数和16 位地址长度进行译码的，即使是那些用bits 32 编译的指令。进入保护模式后，受CS 段描述符高速缓存器中实模式残留内容的影响，处理器进入16 位保护模式工作。如果保护模式下的代码是16 位的，影响可能不大，但如果是用bits 32 编译的，那么，由于对对操作数和默认地址大小的解释不同，指令的执行结果可能会不正确，所以必须清空流水线。同时，那些通过乱序执行得到的中间结果也是无效的，必须清理掉，让处理器串行化执行，即，重新按指令的自然顺序执行。

怎么办呢？这里有一个两全其美的方案，那就是使用远转移指令jmp 或者远过程调用指令call。处理器最怕转移指令，遇到这种指令，一般会清空流水线，并串行化执行；另一方面，远转移会重新加载段选择器CS，并刷新描述符高速缓存器中的内容。一个建议的方法是在设置了控制寄存器CR0 的PE 位之后，立即用jmp 或者call 转移到当前指令流的下一条指令上。为此，代码清单11-1 第51 行，用32 位远转移指令来转移到紧挨着当前指令的下一条指令：

```
        jmp dword 0x0008:flush
```

这条指令和位于它前面的指令一样，默认地是用“bits 16”编译的，而且使用了关键字“dword”，该关键字修饰偏移地址，意思是要求使用32 位的偏移量。因此，会有指令前缀0x66，编译之后的结果是

```
66 EA 80 00 00 00 08 00
```

实际上，因为处理器现在处于16 位保护模式，说到底还是16 位模式，因此，使用以下的16位远转移指令可能更自然一些：

```
jmp 0x0008:flush
```

如果使用这条指令，那么，同样用“bits 16”编译，生成的机器指令是

```
EA 7B 00 08 00
```

16 位的绝对远转移指令只有5 个字节，使用16 位的偏移量。因此，它会使标号flush 的汇编地址相应地变小，变成0x007B，而不是从前的0x0080。

16 位保护模式是从80286 处理器开始引入的，80286 没有对8086 的寄存器进行扩展，依然使用AX、BX、CX、DX、SI、DI、BP、SP，等等。所以，在16 位保护模式下，段可以起始于任何地方，但每个段的最大长度是64KB，只使用16 位的偏移量。16 位保护模式并不重要，为了绕开它，我们才在这里使用了32 位的远转移指令。

注意，不管你用的是16 位远转移，还是32 位远转移，因为现在已经处于保护模式下，处理器都将把第一个操作数0x0008 视为段选择子，而不是实模式下的逻辑段地址。

因为处理器实际上是在保护模式下执行该指令的，因此，它会重新解释这条指令的含义。我们知道，操作数的默认大小（16 位还是32 位）是由描述符的D 位决定的，确切地说，是由段寄存器的描述符高速缓存器中的D 位决定的，毕竟，要访问一个段，必须首先将它的描述符传送到段寄存器的描述符高速缓存器中。当它刚进入保护模式时，CS 的描述符高速缓存器依然保留着实模式时的内容，其D 位是“0”，因此，在那个时刻，处理器运行在16 位保护模式下。

因为处理器已经进入保护模式，所以，0x0008 不再是逻辑段地址，而是保护模式下的段描述符选择子。在前面定义GDT 的时候，它的第2个（1 号）描述符对应着保护模式下的代码段。因此，其选择子为

0x0008（索引号为1，TI 位是0，RPL 为00）。当指令执行时，处理器加载段选择器CS，从GDT 中取出相应的描述符加载到CS 描述符高速缓存。

保护模式下的代码段，基地址为0x00007c00，段界限为0x1ff，长度为0x200，正好对应着当前程序在内存中的区域。在这种情况下，上面那条指令执行时，目标位置在段内的偏移量就是标号flush 的汇编地址，处理器用它的数值来代替指令指针寄存器EIP 的原有内容。

在16 位保护模式下执行带前缀0x66 的指令，那么，很好，处理器会按32 位的方式执行，使用32 位的偏移量。于是，它将0x0008 加载到CS 选择器，并从GDT 中取出对应的描述符，加载CS 描述符高速缓存器；同时，把指令中给出的32 位偏移量传送到指令指针寄存器EIP。很自然地，处理器就从新的位置开始取指令执行了。

可以在Bochs 中观察段寄存器在各个阶段的状态，包括计算机加电后、设置CR0 寄存器的PE位后和执行JMP 指令后的状态。具体方法请参见本章11.9.2、11.9.3 和11.9.4 节。通过了解这些状态变化，可以进一步加深对处理器如何进入保护模式的理解。

从进入保护模式开始，之后的指令都应当是按32 位操作数方式编译的。因此，第53 行，使用了伪指令[bits 32]。当处理器执行到这里时，它会按32 位模式进行译码，这正是我们所希望的。

代码清单11-1 第56～74 行，用于把描述符选择子0x10 加载到段选择器DS，并自动加载描述符高速缓存器。因为该数据段实际上是文本模式下的显示缓冲区，故大部分指令都用于在屏幕上显示字符串“Protect mode OK”。保护模式下的数据段访问已经在上一节里讨论过了，这里不再赘述。另外，处理器模式的变化对外围设备没有影响，它们是无法感知的，而且只按自己的方式工作。

注意，在保护模式下，不允许使用mov 指令改变段寄存器CS 的内容，比如：

```
mov cs,ax
```

企图这样做将导致处理器产生一个无效操作码的异常中断。

11.8 保护模式下的栈

11.8.1 关于栈段描述符中的界限值

第77～79 行用于初始化保护模式下的栈。栈段描述符是GDT 中的第4 个（3 号）描述符，栈的32 位线性基地址是0x00000000，段界限为0x07a00，粒度为字节，属于可读可写、向下扩展的数据段。

栈是向下扩展的，因此，描述符中的段界限，和向上扩展的段含义不同。对于向上扩展的段，段内偏移量是从0 开始递增，偏移量的最大值是界限值和粒度的乘积；而对于向下扩展的段来说，因为它经常用做栈段，而栈是从高地址向低地址方向推进的，故段内偏移量的最小值是界限值和粒度的乘积加一。在32 位代码中，是用ESP 作为栈指针的。因此，这里的段界限，用来和段粒度一起，决定ESP 寄存器所能具有的最小值。即，栈操作时，必须符合条件：

```
ESP ＞ 段界限×粒度值
```

对于描述符中G 位是“0”的段来说，粒度值是1（字节）；而对于G 位是“1”的段来说，粒度值是4096（4KB）。

在当前代码中，ESP 寄存器的内容被初始化为0x00007c00。假如此时执行以下指令：

```
push edx
```

那么，因为要压入一个32 位数，所以处理器先将ESP 的内容减去4，再压入数据。此时，ESP 寄存器的内容为（扩展到32 位）：

```
0x00007C00－4＝0x00007BFC
```

在当前栈段的描述符中，段界限为0x07a00，粒度是字节，故作为栈的界限，实际使用的数值是（扩展到32 位）：

```
0x07A00×1＝0x00007A00
```

对于栈段来说，段界限的值加一，就是段内偏移量的最小值。因为要访问的段内偏移量0x00007BFC 大于实际使用的段界限值0x00007A00，故处理器允许执行该操作，并用描述符高速缓存中的32位基地址0x00000000 加上这里的偏移量0x00007BFC，共同形成32 位线性地址访问栈，将寄存器EDX 的内容压入。否则，处理器阻止当前操作，引发一个异常中断。

你可能觉得当前的栈段很完美。但不得不说，这是一个非常糟糕的栈定义。结合本章的程序，很明显，我们的本意是要定义一个只有512字节的栈空间，从物理地址0x00007A00 开始，到物理地址0x00007C00结束，如图11-14 所示。

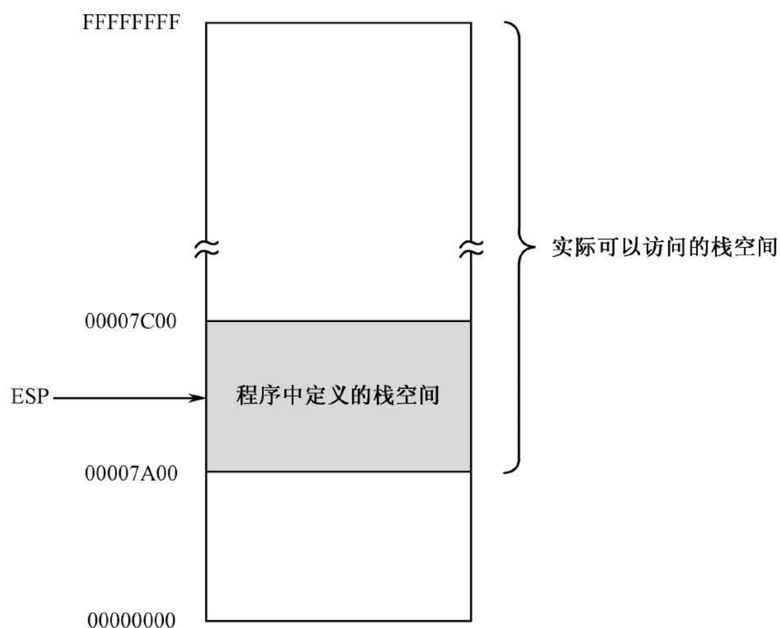


图11-14　栈段的界限和栈的安全访问

尽管我们的本意是定义一个只有512 个字节的栈，但是，从该段的描述符来看，这个段的空间却是非常巨大的。假如一切正常，特别是指令执行正常，那不会有什么问题。但是，在程序失控的情况下，ESP 的内容可能会是任何预料不到的值，比如0xFFFFFFFF。即使是这样，它也是合法的值，毕竟它大于0x00007A00。因为当前栈段的线性基地址为0x00000000，所以，实际可以访问的空间是从物理地址0xFFFFFFFF 到0x00007A00。显然，这超出了我们的预期。在下一章里，我们将继续讨论如何用更好的方法来创建栈。

11.8.2 检验32 位下的栈操作

代码清单11-1 中，最后的指令用于演示保护模式下的栈操作。我们已经知道，对于存储器的段来说，其描述符的D/B 位，对于代码段来说，是D 位；对于栈段来说，是B 位。

隐式的栈操作（push、pop、call、ret 和iret）涉及两个段：一个是指令所在的代码段；另一个是指令执行时，所使用的栈段。正如上一章所述，16 位下的栈操作，其默认的操作数大小是16 位的，而且使用的栈指针寄存器是SP；32 位下的隐式栈操作，其默认的操作数大小是32 位的，使用ESP 寄存器。

在本章里，当前程序的代码段，其描述符的D 位是“1”，所以，当进行隐式的栈操作时，默认地，每次压栈操作时，压入的是双字；当前程序所使用的栈段，其符述符的B 位也是“1”，默认地，使用栈指针寄存器ESP 进行操作。为此，从第81 行开始的指令用于检验这个事实。

因此，第81 行，先保存当前栈指针的内容到EBP 寄存器；接着，第82 行，向栈中压入立即数。该立即数为字符“.”的ASCII 码，这个值是在编译阶段计算的。

因为当前正在执行的代码段是32 位的，其描述符的D 位是“1”，故push 指令默认的操作数大小是32 位。正如在上一章里所讲的，关键字“byte”仅仅是给编译器用的，告诉它，该指令对应的格式为push imm8，必须使用操作码0x6A，而不是用来在编译后的机器指令前添加指令前缀。因此，该指令实际在处理器上执行时，压入栈中的是一个双字，也就是4 字节，高24 位是该字节符号的扩展。

当前指令执行时，所访问的栈，其描述符的B 位也是“1”，故处理器在进行栈操作时，用的是32 位栈指针寄存器ESP。它首先将ESP 的内容减去4，再写入数值，数据保存的位置是SS:ESP。

现在，理论上，将EBP 的内容减去4 之后，应该和ESP 的内容相同。为了证实这一点，第84～86 行，将原先保存的EBP 内容减去4，再和现行的ESP 比较，看是否相等。如果相等，则立即将刚才压入的字符出栈，并显示在前面的字符串后。不存在从栈中弹出字节的指令，因为名义上可以压入字节，但实际上它们是作为16 位或者32 位有符号数压入的。

当然，如果经过验证，EBP 和ESP 不相等，那么，将不会显示句点，直接转移到程序的最后，执行停机指令。因为现在已经禁止了中断，故除了NMI，没有任何原因会导致处理器被激活。

检测点11.2

1． 用Bochs 调试本章的程序，在第84 行，即sub ebp,4 处设置断点，观察栈的状态。此时，栈顶的双字数据是（　　　　）。

2．以下说法，哪些是正确的（可多选）？

A．在x86 处理器的实模式下，可以在栈中压入16 位或者32 位数据。

B．在x86 处理器的32 位保护模式下，可以在栈中压入16 位或者32 位数据。

C．在x86 处理器的32 位保护模式下，如果栈段描述符的B 位是0，则使用SP 寄存器。压入16 位数据时，是SP 先减去2；压入32 位数据时，是SP 先减去4。

D．在x86 处理器的32 位保护模式下，如果栈段描述符的B 位是1，则使用ESP 寄存器。压入16 位数据时，是ESP 先减去2；压入32 位数据时，是ESP 先减去4。

E．在x86 处理器的32 位保护模式下，只能在栈中压入32 位数据。

11.9 程序的运行和调试

11.9.1 运行程序并观察结果

编译代码清单11-1，生成二进制文件c11_mbr.bin。用FixVhdWr 工具将此文件写入虚拟硬盘的主引导扇区，然后启动虚拟机，如果没有问题的话，显示的结果应当如图11-15 所示。

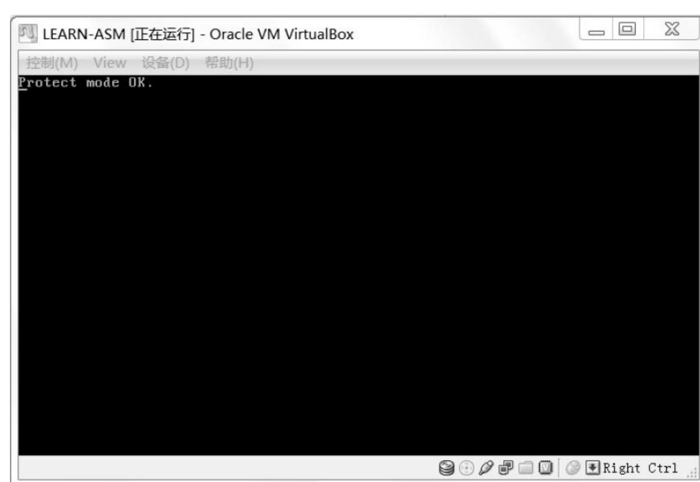


图11-15 本章程序的运行结果

11.9.2 处理器刚加电时的段寄存器状态

在x86 处理器加电后，它的固件会对自身进行初始化，可选地，还可以执行一个内置的自测试（Build-In Self-Test，BIST）。如果执行了BIST，那么，当测试通过后，EAX 寄存器被清零，否则，EAX 的内容为非零。如果不执行BIST，那么，EAX 寄存器的内容默认地也是0。在这些工作完成后，才开始取指令和执行指令。

不管怎样，当处理器初始化完成后，它内部的各个寄存器，包括通用寄存器、段寄存器、控制寄存器、指令指针寄存器EIP、栈指针寄存器ESP，以及我们尚未接触过的其他寄存器，都会有一个预置的值。至于它们的初始值是什么，可以查阅相关资料，比如INTEL 公司的手册Intel® 64 and IA-32 Architectures Software Developer’s Manual，它和本书一

起，是你案头必备的资料（网上有大量的下载链接）。当然，如果不想查阅手册，Bochs 也能帮上你的忙。

Bochs 是用软件来模拟处理器的工作，所以它有这个能力。要想知道处理器加电后，各个寄存器都预置了什么内容，可以选择在它执行第一条指令之前，使用调试命令来显示它们。比如，可以用“r”命令显示各个通用寄存器的初始内容。当然，我们现在只想知道各个段寄存器中都有些什么。

如图11-16 所示，在处理器开始执行它本次加电以来的第一条指令前，可以用“sreg”命令察看各个段寄存器此时的状态。显然，段寄存器CS 的内容是0xF000，而其他段寄存器都是0。

```
Bochs for Windows - Console
00000000000i[     ] reset of 'serial' plugin device by virtual method
00000000000i[     ] reset of 'gameport' plugin device by virtual method
00000000000i[     ] reset of 'iodebug' plugin device by virtual method
00000000000i[     ] set SIGINT handler to bx_debug_ctrlc_handler
Next at t=0
(0) [0x00000000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b        ; ea5b
e000f0
<bochs:1> sreg
es:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
cs:0xf000, dh=0xff0093ff, dl=0x0000ffff, valid=7
        Data segment, base=0xffff0000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
fs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
gs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x0000000000000000, limit=0xffff
idtr:base=0x0000000000000000, limit=0xffff
<bochs:2> _
```

图11-16　x86 处理器加电后的段寄存器状态

图中还显示了段寄存器描述符高速缓存器的内容，这些内容也是加电之后预置的。首先，“dh”是段描述符的高32 位；“dl”是段描述符的低32 位。因为是加电预置的内容，并非来自于描述符表，所以，“dh”和“dl”的内容是Bochs 根据段寄存器描述符高速缓存器的内容构造的。

与此同时，Bochs 还根据各个段寄存器描述符高速缓存器的内容，给出了摘要信息。其中，“ Data segment ”表示该段是数据段；“ base ”指示段的基地址；“ limit ”指示段的界限；“Read/Write”表示段可读可写；“Accessed”指示段曾经被访问过。

8086 处理器访问内存时，是把16 位段寄存器的内容左移4 位，加上16 位偏移地址，比如

```
mov ax,[0x06]
```

在32 位处理器上，每次向段寄存器传送逻辑段地址时，处理器即在段寄存器描述符高速缓存器中存放一个左移后的20 位基地址。一个典型的例子是

```
mov ds,ax
```

因此，即使在实模式下，处理器也是用段寄存器描述符高速缓存器的32 位基地址加上16 位偏移地址访问内存，只不过基地址的高12 位通常是0。当然，也有一个例外，那就是在处理器刚电加时，CS 段描述符高速缓存器中的基地址被预置为0xFFFF0000，这使得处理器取第一条指令时，地址线的高位部分被强制为“1”。又因为加电后，EIP 的预置内容是 0x0000FFF0，故，处理器第一次取指令时发出的地址是0xFFFFFFF0。之所以这样做，是因为处理器的设计者希望把ROM-BIOS 放到4GB 可寻址内存范围的最高端，这样，4GB 以下，连同传统的低端1MB 都是连续的RAM 区，连续的、不间断的RAM 能为操作系统管理内存带来方便。

问题在于，计算机制造商们会考虑很多现实问题。老的硬件和软件依赖于低端1MB 的ROMBIOS 来工作，这涉及到兼容性。最终，这两个地址区段都指向同一块ROM 芯片。

从图中可以看到，即将执行的第一条指令是jmp far f000:e05b（jmp 0xf000:0xe05b）。当这条指令执行后，处理器用0xF000 的值左移4位，存放到段寄存器描述符高速缓存器。于是，处理器地址线的高位部分不再为“1”，这又转到低地址端的BIOS 执行了。如图11-17 所示，当执行远转移指令后，CS 描述符高速缓存器中的基地址变为0x000F0000，下一条指令xor ax,ax 的物理地址是0x000FE05B。

```
Bochs for Windows - Console
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x0000000000000000, limit=0xffff
idtr:base=0x0000000000000000, limit=0xffff
<bochs:2> s
Next at t=1
(0) [0x00000000000fe05b] f000:e05b (unk. ctxt): xor ax, ax             ; 31c0

<bochs:3> sreg
es:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
cs:0xf000, dh=0x0000930f, dl=0x0000ffff, valid=1
        Data segment, base=0x000f0000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
fs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
gs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x0000000000000000, limit=0xffff
idtr:base=0x0000000000000000, limit=0xffff
<bochs:4>
```

图11-17　处理器加电并执行第一个远转移指令后的段寄存器内容

图中还显示了全局描述符表寄存器GDTR 的内容。很显然，GDT 的基地址是0，表界限是0xFFFF。

注意，在进入主引导程序时，这些段寄存器的内容（包括GDTR）和处理器刚加电时不再相同。原因很简单，BIOS 的加电自检程序在执行期间要进入保护模式进行测试，这将改变相关段寄存器的内容。

11.9.3　设置PE 位后的段寄存器状态

一旦设置了控制寄存器CR0 的PE 位，处理器就进入了保护模式。但是，除非我们主动刷新段寄存器的内容，否则，段寄存器依然保持实模式下的内容不变。这就是说，设置了PE 位之后，刚进入保护模式时的段寄存器内容，就是实模式下的段寄存器内容。

如图11-18 所示，我们在执行mov cr0,eax 指令之后立即显示各个段寄存器的内容。实际上，尽管进入了保护模式，但显示的依然是实模式的内容。

从图中可以看出，代码段寄存器CS 描述符高速缓存器的dh 为0x00009300，即，G=0，D=0，L=0，P=1，DPL=00，S=1，TYPE=0011。通俗地说，这是一个粒度为字节的数据段。在实模式下，这些属性信息是没有作用的，定义成数据段也无所谓。

一旦设置了CR0 寄存器的PE 位，进入了保护模式，那么，理论上，这些属性信息就变得有用了，有意义了，但同时也是无效的。原因

在于，它应当是一个代码段，而不是数据段。

```
Bochs for Windows - Console
Next at t=17825058
(0) [0x0000000000007c73] 0000:7c73 (unk. ctxt): mov cr0, eax            ; 0f22
c0
<bochs:29> s
Next at t=17825059
(0) [0x0000000000007c76] 0000:0000000000007c76 (unk. ctxt): jmp far 0008:0000007
e       ; 66ea7e0000000800
<bochs:30> sreg
es:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
cs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ds:0x07e0, dh=0x00009300, dl=0x7e00ffff, valid=7
        Data segment, base=0x00007e00, limit=0x0000ffff, Read/Write, Accessed
fs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
gs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x0000000000007e00, limit=0x1f
idtr:base=0x0000000000000000, limit=0x3ff
<bochs:31>
```

图11-18　刚进入保护模式时的段寄存器状态

尽管如此，这对处理器继续取指令和执行指令没有影响。原因是，在保护模式下，对描述符有效与否的检查，通常只在加载段寄存器（选择器），并刷新描述符高速缓存器的时候进行。对代码段来说，典型的例子是远转移或者远过程调用，比如

```
jmp 0x0008:0x0002
```

而对于数据段来说，典型的例子是加载段选择子，比如

```
mov ds,ax
```

当这类指令执行时，处理器用指令中给出的选择子找到描述符，如果描述符有效，就将选择子加载到段寄存器（选择器），并把描述符加载到描述符高速缓存器。因此，一个不合格的、无效的描述符不可能被加载到段寄存器的描述符高速缓存器。不过，当前这个情况比较特殊，因为它是进入保护模式前遗留下来的。

11.9.4　JMP 指令执行后的段寄存器状态

x86 处理器的保护模式分为两种：16 位保护模式和32 位保护模式。处理器加电、设置CR0寄存器的PE 位后，CS 描述符高速缓存器中的D

位是0，处理器根据此位将自己的状态设置为16位保护模式。

16 位保护模式和实模式相比，在指令格式和寻址方式上是相同的。因此，如果进入保护模式后的指令是16 位指令，从理论上来说不会有什么问题。实际上，因为各个段寄存器，特别是数据段寄存器中的内容并没有更新，不适当的内存访问将导致不可预知的问题。

以上所说，是解释为什么进入保护模式后，处理器还能接着往下执行的原因。问题在于，它虽然还能按原来的执行流程进行，但后面的代码是用bits 32 编译的。当处理器处在16 位保护模式时，它会按16 位的方式译码32 位指令，这是不行的。

因此，我们使用了jmp dword 0x0008:flush 指令。这是一个默认地用bits 16 编译，在16 位模式下译码，但要求按32 位执行的指令。所以，它必然具有反转操作数大小的前缀0x66。

```
Bochs for Windows - Console
(0) [0x0000000000007c76] 0000:0000000000007c76 (unk. ctxt): jmp far 0008:0000007
e      ; 66ea7e0000000800
<bochs:14> s
Next at t=17825060
(0) [0x0000000000007c7e] 0008:000000000000007e (unk. ctxt): mov cx, 0x0010
       ; 66b91000
<bochs:15> sreg
es:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
cs:0x0008, dh=0x00409900, dl=0x7c0001ff, valid=1
        Code segment, base=0x00007c00, limit=0x000001ff, Execute-Only, Non-Confo
rming, Accessed, 32-bit
ss:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ds:0x07e0, dh=0x00009300, dl=0x7e00ffff, valid=7
        Data segment, base=0x00007e00, limit=0x0000ffff, Read/Write, Accessed
fs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
gs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x0000000000007e00, limit=0x1f
idtr:base=0x0000000000000000, limit=0x3ff
<bochs:16>
```

图11-19　执行JMP 指令后的段寄存器内容

如图11-19 所示，这条远转移指令的机器码是66 EA 7E 00 00 00 08 00，在它执行后，我们又一次显示了段寄存器的状态。从图中可以看出，段寄存器CS 及其描述符高速缓存器的内容都已更新为有效内容。此后，处理器根据描述符高速缓存器的D 位（此时为“1”），把自己设置为32 位模式。

11.9.5　察看全局描述符表GDT

可以察看全局描述符表的内容，但前提是必须加载了全局描述符表寄存器GDTR 才行，因为Bochs 要先访问GDTR 取得GDT 的基地址和界限信息。

如图11-20 所示，单步执行本章的程序，在执行指令lgdt [cs:gdt_size+0x7c00]后停下来，输入“info gdt”命令，来显示GDTR 的内容。

```
Bochs for Windows - Console
(0) [0x0000000000007c50] 0000:7c50 (unk. ctxt): mov dword ptr ds:[bx+28], 0x0040
9600 ; 66c7471c00964000
<bochs:19> s
Next at t=17825050
(0) [0x0000000000007c58] 0000:7c58 (unk. ctxt): mov word ptr cs:0x7d0a, 0x001f ;
 2ec7060a7d1f00
<bochs:20> s
Next at t=17825051
(0) [0x0000000000007c5f] 0000:7c5f (unk. ctxt): lgdt cs:0x7d0a           ; 2e0f
01160a7d
<bochs:21> s
Next at t=17825052
(0) [0x0000000000007c65] 0000:7c65 (unk. ctxt): in al, 0x92             ; e492

<bochs:22> info gdt
Global Descriptor Table (base=0x0000000000007e00, limit=31):
GDT[0x00]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x01]=Code segment, base=0x00007c00, limit=0x000001ff, Execute-Only, Non-Con
forming, 32-bit
GDT[0x02]=Data segment, base=0x000b8000, limit=0x0000ffff, Read/Write
GDT[0x03]=Data segment, base=0x00000000, limit=0x00007a00, Read/Write, Expand-do
wn
You can list individual entries with 'info gdt [NUM]' or groups with 'info gdt [
NUM] [NUM]'
<bochs:23> _
```

图11-20　察看全局描述符表GDT 的内容

如图中所示，Bochs 给出了每个描述符的索引和相关信息。比如，它用“GDT[0x01]”表示GDT 的1 号槽位；“Code segment”表示代码段；“base”表示段的32 位基地址；“limit”表示段界限值；“Execute-Only”表示只执行；“Non-Conforming”表示非依从的；“32-bit”表示这是一个32 位的段。

11.9.6　察看控制寄存器的内容

本章在进入保护模式前，需要设置控制寄存器CR0 的PE 位。在往后的学习过程中，有可能要察看控制寄存器的状态，这里简单做一下介绍。

```
Bochs for Windows - Console
Next at t=17825055
(0) [0x0000000000007c6b] 0000:7c6b (unk. ctxt): cli                    ; fa
<bochs:25> s
Next at t=17825056
(0) [0x0000000000007c6c] 0000:7c6c (unk. ctxt): mov eax, cr0           ; 0f20
c0
<bochs:26> s
Next at t=17825057
(0) [0x0000000000007c6f] 0000:7c6f (unk. ctxt): or eax, 0x00000001     ; 6683
c801
<bochs:27> s
Next at t=17825058
(0) [0x0000000000007c73] 0000:7c73 (unk. ctxt): mov cr0, eax           ; 0f22
c0
<bochs:28> creg
CR0=0x60000010: pg CD NW ac wp ne ET ts em mp pe
CR2=page fault laddr=0x0000000000000000
CR3=0x0000000000000000
    PCD=page-level cache disable=0
    PWT=page-level write-through=0
CR4=0x00000000: smep osxsave pcid fsgsbase smx vmx osxmmexcpt osfxsr pce pge mce
 pae pse de tsd pvi vme
CR8: 0x0
EFER=0x00000000: ffxsr nxe lma lme sce
<bochs:29>
```

图11-21　察看控制寄存器的内容

如图11-21 所示，可以用命令“creg”（control register）来察看控制寄存器的内容。32 位处理器有多个控制寄存器，不单单是CR0。所以，它会显示所有控制寄存器的内容。如图中所示，CR0 的内容是0x60000010，为了方便起见，Bochs 给出了各个控制位的状态，小写表示该位是“0”，大写表示该位是“1”，即处于置位状态。显然，此时PE 位是“0”，处理器并未工作在保护模式下，这是因为指令mov cr0,eax 指令还没有执行。至于其他控制寄存器，以及CR0 其他各控制位的含义，我们将在后面慢慢接触，这里不用理会。

本章习题

在我的计算机上，创建虚拟机时指定的内存容量是64MB。因此，实际有效的物理地址范围是0x00000000～0x03FFFFFF。我发现，如果将代码清单11-1 的第79 行改成以下指令，程序工作正常，能显示句点：

```
mov esp,0x04000003
```

但是，如果改成

```
mov esp,0x04000004
```

就不能显示句点。请问这是什么原因。注意，处理器在访问内存时，并不检验内存单元的有效性。

第12章　存储器的保护

处理器引入保护模式的目的是提供保护功能，其中很重要的一个方面就是存储器保护。存储器的保护功能可以禁止程序的非法内存访问，比如，向代码段写入数据、访问段界限之外的内存位置等。很多时候，这类问题都是由于编程疏漏引起的，属于有缺陷的软件，但也不排除软件的功能本身就是恶意的。不过，一旦能够及时发现和禁止这些非法操作，在程序失去控制之前引发异常中断，就可以提高软件的可靠性，降低整个计算机系统的安全风险。

凡事都有两面。利用存储器的保护功能，也可以实现一些有价值的功能，比如虚拟内存管理。当处理器访问一个实际上不存在的段时，会引发异常中断。操作系统可以利用这一点，通过接管异常处理过程，并用硬盘来进行段的换入和换出，从而实现在较小的内存空间运行尽可能大、尽可能多的程序。本章的学习目标是：

1．通过实例来认识处理器是如何进行存储器的保护的。

2．了解别名段的意义和作用。

3．以一个字符串排序过程作为例子，演示保护模式下的内存数据访问，体验一下它们与在实模式下访问数据段有什么不同。同时，在这个过程中学习用汇编语言实现冒泡排序算法，以及一条新的x86 处理器指令xchg。

12.1　代码清单12-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：12-1（主引导扇区程序）

源程序文件：c12_mbr.asm

12.2　进入32 位保护模式

12.2.1　话说mov ds,ax 和mov ds,eax

本章的代码和上一章有几分类似，但实质上有很大区别。

我们知道，段寄存器（选择器）的值只能用内存单元或者通用寄存器来传送，一般的指令格式为

```
mov sreg,r/m16
```

这里有一个常见的例子：

```
mov ds,ax
```

在16 位模式下，传送到DS 中的值是逻辑段地址；在32 位保护模式下，传送的是段描述符的选择子。无论传送的是什么，这都不重要，重要的是，在16 位模式和32 位模式下，一些老式的编译器会生成不同的机器代码。下面是一个例证：

```
[bits 16]
mov ds,ax         ;8E D8

[bits 32]
mov ds,ax         ;66 8E D8
```

由于在16 位模式下，默认的操作数大小是字（2 字节），故生成8E D8 也不难理解。在32 位模式下，默认的操作数大小是双字（4 字节）。由于指令中的源操作数是16 位的AX，故编译后的机器码前面应当添加前缀0x66 以反转默认的操作数大小，即66 8E D8。

很遗憾，由于这一点点区别，有前缀的和没有前缀的相比，处理器在执行时会多花一个额外的时钟周期。问题在于，这样的指令用得很频繁，而且牵扯到内存段的访问，自然也很重要。因此，它们在16 位模式和32 位模式下的机器指令被设计为相同。即都是8E D8，不需要指令前缀。

这可难倒了很多编译器，它们固执地认为，在32 位模式下，源操作数是16 位的寄存器AX时，应当添加指令前缀。好吧，为了照顾它们，很多程序员习惯使用这种看起来有点别扭的形式：

```
mov ds,eax
```

你别说，还真有效，果然生成的是不加前缀的8E D8。

说到这里，我觉得NASM 编译器还是非常优秀的，起码它不会有这样的问题。因此，不管处理器模式如何变化，也不管指令形式如何变化，以下代码编译后的结果都一模一样：

```
[bits 16]
mov ds,ax          ;8E D8
mov ds,eax         ;8E D8

[bits 32]
mov ds,ax          ;8E D8
mov ds,eax         ;8E D8
```

和这个示例一样，其他从通用寄存器到段寄存器的传送也符合这样的编译规则。因此，代码清单12-1 第7、8 行，用于通过寄存器EAX 来初始化栈段寄存器SS。

12.2.2 创建GDT 并安装段描述符

准备进入保护模式。

首先是创建GDT，并安装刚进入保护模式时就要使用的描述符。第12～15 行，首先计算GDT 在实模式下的逻辑地址。在上一章里，GDT 的大小和线性基地址分别是用两个标号gdt_size 和gdt_base 声明和初始化的：

```
gdt_size dw 0
gdt_base dd 0x00007e00
```

但是，如后面的第107、108 行所示，现在已经改成

```
pdgt dw 0
     dd 0x00007e00
```

另外一个区别是计算GDT 逻辑地址的方法。在32 位处理器上，即使是在实模式下，也可以使用32 位寄存器。所以，第12 行，直接将GDT 的32 位线性基地址传送到寄存器EAX 中。

我们知道，32 位处理器可以执行以下除法操作：

```
div r/m32
```

其中，64 位的被除数在EDX:EAX 中，32 位被除数可以在32 位通用寄存器中，也可以在32位内存单元中。因此，第13～15 行，用64 位的被除数EDX:EAX 除以32 位的除数EBX。指令执行后，EAX 中的商是段地址，仅低16 位有效；EDX 中的余数是段内偏移地址，仅低16 位有效。

第17、18 行，初始化段寄存器DS，使其指向GDT 所在的逻辑段。

第21、22 行，安装空描述符。该描述符的槽位号是0，处理器不允许访问这个描述符，任何时候，使用索引字段为0 的选择子来访问该描述符，都会被处理器阻止，并引发异常中断。在现实中，一个忘了初始化的指针往往默认值就是0，所以空描述符的用意就是阻止不安全的访问。很多人喜欢用这个槽位来记载一些私人信息，做一些特殊的用途，认为反正处理器也不用它。但是，这样做可能是不安全的，还没有证据表明Intel 公司保证决不会使用这个槽位。

第25、26 行，安装保护模式下的数据段描述符。参考前面的段描述符格式，可以看出，该段的线性基地址位于整个内存的最低端，为0x00000000；属于32 位的段，段界限是0xFFFFF。但是要注意，段的粒度是以4KB 为单位的。对于以4KB（十进制数4096 或者十六进制数0x1000）为粒度的段，描述符中的界限值加1，就是该段有多少个4KB。因此，其实际使用的段界限为

```
（描述符中的段界限值＋1）×0x1000－1
```

将其展开后，即

```
描述符中的段界限值×0x1000＋0x1000－1
```

因此，在换算成实际使用的段界限时，其公式为

```
描述符中的段界限值×0x1000＋0xFFF
```

这就是说，实际使用的段界限是

```
0xFFFFF×0x1000＋0xFFF＝0xFFFFFFFF
```

也就是4GB。就32 位处理器来说，这个地址范围已经最大了。一旦使用这个段，就可以访问0 到4GB 空间内的任意一个单元，这是本书开篇以来，从来没有过的事情。

第29、30 行，安装保护模式下的代码段描述符。该段是32 位的代码，线性基地址为0x00007C00；段界限为0x001FF，粒度为字节。对于向上扩展的段来说，段界限在数值上等于段的长度减去1，因此该段的长度是0x200，即512 字节。

根据上一章的经验，该段实际上就是当前程序所在的段（正在安装该描述符呢），也就是主引导程序所在的区域。尽管在描述符中把它定义成32 位的段，但它实际上既包含16 位代码，也包含32 位代码。[bits 32]之前的代码是16 位的，之后的代码是32 位的。不过，在该描述符生效的时候，处理器的执行流已经位于32 位代码中了。

第33、34 行，安装保护模式下的数据段描述符。该段是32 位的数据段，线性基地址为0x00007C00；段界限为0x001FF，粒度为字节。可以看出，该描述符和前面的代码段描述符，描述和指向的是同一个段。你可能很想知道，这样做的用意何在？

参见上一章的表11-1，我们都已经知道，在保护模式下，代码段是不可写入的。所谓不可写入，并非是说改变了内存的物理性质，使得内存写不进去，而是说，通过该段的描述符来访问这个区域时，处理器不允许向里面写入数据或者更改数据。

但是，很多时候，又需要对代码段做一些修改。比如在调试程序时，需要加入断点指令int3。不管怎么样，如果需要访问代码段内的数据，只能重新为该段安装一个新的描述符，并将其定义为可读可写的数据段。这样，当需要修改代码段内的数据时，可以通过这个新的描述符来进行。

像这样，当两个以上的描述符都描述和指向同一个段时，把另外的描述符称为别名（alias）。注意，别名技术并非仅仅用于读写代码段，如果两个程序想共享同一个内存区域，可以分别为每个程序都创建一个描述符，而且它们都指向同一个内存段，这也是别名应用的例子。

第36、37 行，安装保护模式下的栈段描述符。该段的线性基地址是0x00007C00，段界限为0xFFFFE，粒度为4KB。

尽管该段和代码段使用同一个线性基地址，但这不会有什么问题，代码段是向上（高地址方向）扩展的，而栈段是向下（低地址方向）扩展的。至于段界限为0xFFFFE，粒度为4KB，我知道你可能会有某些疑问，这些事情马上就会讲到。

第40 行，设置GDT 的界限值为39，因为这里共有5 个描述符，总大小为40 字节，界限值为39。后面的代码用于进入保护模式，差不多和上一章相同，不再赘述。GDT 和GDT 内的描述符，以及本章程序，它们在内存中的映象如图12-1 所示。



图12-1　本章程序中各个部分在内存中的映象

12.3　修改段寄存器时的保护

随着程序的执行，经常要对段寄存器进行修改。此时，处理器在变更段寄存器以及隐藏的描述符高速缓存器的内容时，要检查其代入值的合法性。

代码清单12-1 第55 行，这是一条直接远转移指令：

```
jmp dword 0x0010:flush
```

这条指令会隐式地修改段寄存器CS。

同样要修改段寄存器的指令还出现在第59～68 行（以下粗体部分）：

```
mov eax,0x0018
mov ds,eax

mov eax,0x0008          ;加载数据段（0:4GB）选择子
mov es,eax
mov fs,eax
mov gs,eax
```

```
mov eax,0x0020          ;0000 0000 0010 0000
mov ss,eax
```

以上的指令涉及所有段寄存器，当这些指令执行时，处理器把指令中给出的选择子传送到段寄存器的选择器部分。但是，处理器的固件在完成传送之前，要确认选择子是正确的，并且该选择子选择的描述符也是正确的。

在当前程序中，选择子的TI 位都是0，故所有的描述符都在GDT中。如图12-2 所示，GDT的基地址和界限，都在寄存器GDTR 中。描述符在内存中的地址，是用索引号乘以8，再和描述符表的线性基地址相加得到的，而这个地址必须在描述符表的地址范围内。换句话说，索引号乘以8 得到的数值，必须位于描述符表的边界范围之内。换句话说，处理

器从GDT 中取某个描述符时，就要求描述符的8 个字节都在GDT 边界之内，也就是索引号×8＋7 小于等于边界。

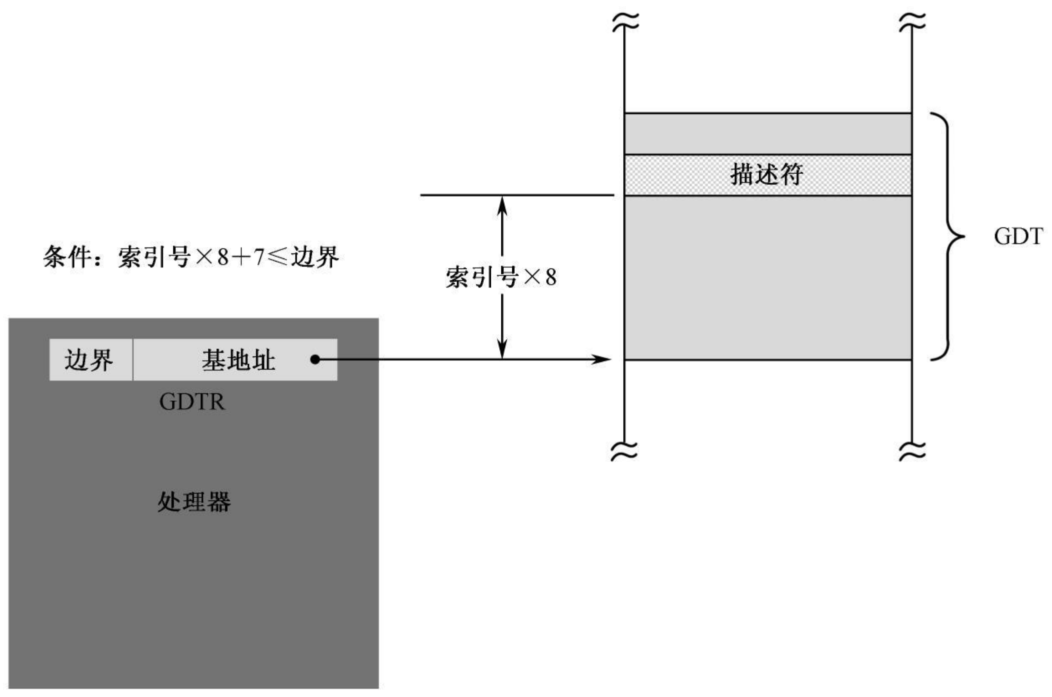


图12-2　索引号的检查

如果检查到指定的段描述符，其位置超过表的边界时，处理器中止处理，产生异常中断13，同时段寄存器中的原值不变。

以上仅仅是检查的第一步。要是通过了上述检查，并从表中取得描述符后，紧接着还要对描述符的类别进行确认。举个例子来说，若描述符的类别是只执行的代码段（表11-1），则不允许加载到除CS 之外的其他段寄存器中。

具体地说，首先，描述符的类别字段必须是有效的值，0000 是无效值的一个例子。

然后，检查描述符的类别是否和段寄存器的用途匹配。其规则如表12-1 所示。

表12-1　段的类别检查

段寄存器	数据段（X=0）		代码段（X=1）	
	只读（W=0）	读写（W=1）	只执行（R=0）	执行、读（R=1）
CS	N	N	Y	Y
DS	Y	Y	N	Y
ES	Y	Y	N	Y
FS	Y	Y	N	Y
GS	Y	Y	N	Y
SS	N	Y	N	N

最后，除了按表12-1 进行段的类别检查外，还要检查描述符中的P位。如果P＝0，表明虽然描述符已被定义，但该段实际上并不存在于物理内存中。此时，处理器中止处理，引发异常中断11。一般来说，应当定义一个中断处理程序，把该描述符所对应的段从硬盘等外部存储器调入内存，然后置P 位。中断返回时，处理器将再次尝试刚才的操作。

如果P＝1，则处理器将描述符加载到段寄存器的描述符高速缓存器，同时置A 位（仅限于当前讨论的存储器的段描述符）。

注意，如表中所指示的那样，可读的代码段类似于ROM。可以用段超越前缀“cs：”来读其中的内容，也可以将它的描述符选择子加载到DS、ES、FS、GS 来做为数据段访问。代码段在任何时候都是不可写的。

一旦上述规则全部验证通过，处理器就将选择子加载到段寄存器的选择器。显然，只有可以写入的数据段才能加载到SS 的选择器，CS 寄存器只允许加载代码段描述符。另外，对于DS、ES、FS 和GS 的选择器，可以向其加载数值为0 的选择子，即

```
xor eax,eax        ;eax = 0
mov ds,eax         ;ds <- 0
```

尽管在加载的时候不会有任何问题，但在，真正要用来访问内存时，就会导致一个异常中断。这是一个特殊的设计，处理器用它来保证系统安全，这在后面会讲到。不过，对于CS 和SS的选择器来说，不允许向其传送为0 的选择子。

继续回到代码清单12-1 中来，第55～68 行的指令执行之后，段寄存器CS 指向512 字节的32位代码段，基地址是0x00007C00；DS 指向512 字节的32 位数据段，该段是上述代码段的别名，因此基地址也是0x00007C00；ES、FS 和GS 指向同一个段，该段是一个4GB 的32 位

数据段，基地址为0x00000000；SS 指向4KB 的32 位栈段，基地址为0x00007C00。

检测点12.1

1．若某段描述符中的段界限是0xFFFFC，当粒度为字节和4KB时，实际使用的段界限是多少？

2．若GDT 的界限为0x87，寄存器AX 的内容为0x0088，则执行指令mov ds,ax 时，处理器会产生异常吗？

12.4　地址变换时的保护

12.4.1　代码段执行时的保护

在32 位模式下，尽管段的信息在描述符表中，但是，一旦相应的描述符被加载到段寄存器的描述符高速缓存器，则处理器取指令和执行指令时，将不再访问描述符表，而是直接使用段寄存器的描述符高速缓存器，从中取得线性基地址，同指令指针寄存器EIP 的内容相加，共同形成32位的物理地址从内存中取得下一条指令。不过，在指令实际开始执行之前，处理器必须检验其存放地址的有效性，以防止执行超出允许范围之外的指令。

每个代码段都有自己的段界限，位于其描述符中。实际使用的段界限，其数值和粒度（G）位有关，如果G＝0，实际使用的段界限就是描述符中记载的段界限；如果G＝1，则实际使用的段界限为

```
描述符中的段界限值×0x1000＋0xFFF
```

该计算公式已经在前面出现过，不再解释。

代码段是向上（高地址方向）扩展的，因此，实际使用的段界限就是当前段内最后一个允许访问的偏移地址。当处理器在该段内取指令执行时，偏移地址由EIP 提供。指令很有可能是跨越边界的，一部分在边界之内，一部分在边界之外，或者一条单字节指令正好位于边界上。因此，要执行的那条指令，其长度减1 后，与EIP 寄存器的值相加，结果必须小于等于实际使用的段界限，否则引发处理器异常。即：

```
0≤（EIP＋指令长度－1）≤实际使用的段界限
```

在本章中，代码段描述符中给出的界限值是0x001FF，粒度是字节，可以认为它就是段内最后一个允许访问的偏移地址。如图12-3 所示，在处理器取得一条指令后，EIP 寄存器的数值加上该指令的长度减1，得到的结果必须小于等于0x000001FF，如果等于或者超出这个数值，必然引发异常中断。

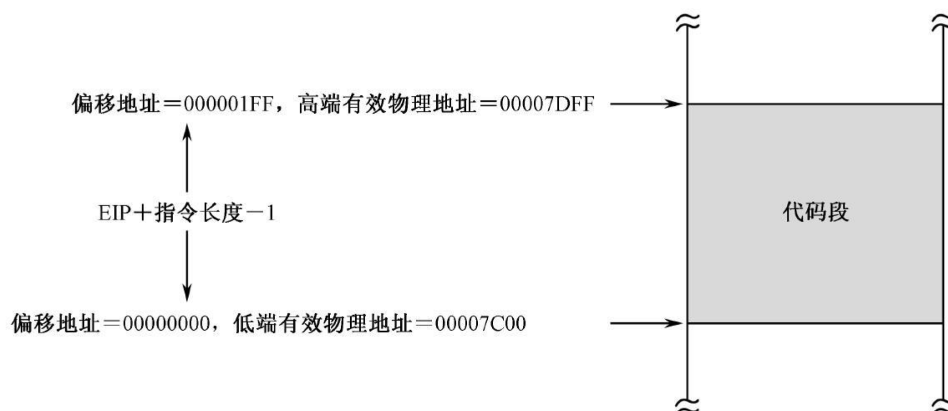


图12-3　对代码段偏移地址的检查

做为一个额外的例子，现在，假设当前代码段的粒度是4KB，那么，因为描述符中的段界限值是0x001FF，故实际使用的段界限是

```
        0x1FF×0x1000+0xFFF=0x001FFFFF
```

可以认为，此数值就是当前段内最后一个允许访问的偏移地址。任何时候，EIP 寄存器的内容加上取得的指令长度减1，都必须小于等于0x001FFFFF，否则将引发处理器异常中断。

任何指令都不允许，也不可能向代码段写入数据。而且，只有在代码段可读的情况下（由其描述符指定），才能由指令读取其内容。

12.4.2　栈操作时的保护

在保护模式下操作时，栈是一个容易令人感到迷惑的话题。在截止到目前的所有例子中，栈段一直是使用向下扩展的内存段，段界限的检查和向上扩展的数据段和代码段不同。当然，栈也可以使用向上扩展的段，即，把数据段用做栈段。在这种情况下，对段界限的检查按数据段的规则进行，但是无论如何，栈本身始终总是向下增长的，即，向低地址方向推进。段的扩展方向用于处理器的界限检查，而对栈的性质以及在栈上进行的操作没有关系。在第16、17 章中，我们会接触到用向上扩展的段作为栈段的情况，现在仍然只讨论向下扩展的栈段。

对栈操作的指令一般是push、pop、ret、iret 等。这些指令在代码段中执行，但实际操作的却是栈段。

现在只讨论32 位的栈段，即，其描述符B 位是1 的栈段。处理器在这样的段上执行压栈和出栈操作时，默认使用ESP 寄存器。

和前面刚刚讨论过的代码段一样，在栈段中，实际使用的段界限也和粒度（G）位相关，如果G＝0，实际使用的段界限就是描述符中记载的段界限；如果G＝1，则实际使用的段界限为

```
描述符中的段界限值×0x1000＋0xFFF
```

栈段是向下扩展的，每当往栈中压入数据时，ESP 的内容要减去操作数的长度。所以，和向高地址方向扩展的段相比，非常重要的一点就是，实际使用的段界限就是段内不允许访问的最低端偏移地址。至于最高端的地址，则没有限制，最大可以是0xFFFFFFFF。也就是说，在进行栈操作时，必须符合以下规则：

```
实际使用的段界限＋1≤（ESP 的内容－操作数的长度）≤0xFFFFFFFF
```

在上一章里，栈段的粒度是字节（G＝0），描述符中的段界限是0x07A00。此时，实际使用的段界限也是0x07A00。

假设现在ESP 的内容是0x00007A04，那么，执行下面的指令时，会怎样呢？

```
push edx
```

因为是要压入一个双字（4 字节），故处理器在向栈中写入数据之前，先将ESP 的内容减去4，得到0x7A00，这就是ESP 寄存器在进行压栈操作时的新值。因为该值小于实际使用的段界限0x7A00 加一（0x7A01），因此不允许执行该操作。

但是，如果执行的是这条指令：

```
push ax
```

那么，因为要压入一个字（2 字节），故实际执行压栈操作时，ESP 的内容是

```
0x7C04－2＝0x7C02
```

结果大于实际使用的段界限加一，允许操作。

回到本章中，看代码清单12-1 第67～69 行。这三行设置栈的线性基地址为0x00007C00，段界限为0xFFFFE，粒度为4KB，并设置栈指针寄存器ESP 的初值为0。

因为段界限的粒度是4KB（G＝1），故实际使用的段界限为

```
0xFFFFE×0x1000＋0xFFF＝0xFFFFEFFF
```

又因为ESP 的最大值是0xFFFFFFFF，因此，如图12-4 所示，在操作该段时，处理器的检查规则是：

```
0xFFFFF000≤（ESP 的内容－操作数的长度）≤0xFFFFFFFF
```

栈指针寄存器ESP 的内容仅仅在访问栈时提供偏移地址，操作数在压入栈时的物理地址要用段寄存器的描述符高速缓存器中的段基址和ESP 的内容相加得到。因此，该栈最低端的有效物理地址是

```
0x00007C00＋0xFFFFF000＝0x00006C00
```

最高端的有效物理地址是

```
0x00007C00＋0xFFFFFFFF＝0x00007BFF
```

也就是说，当前程序所定义的栈空间介于地址为0x00006C00～0x00007BFF 之间，大小是4KB。

现在结合该栈段，用一个实例来说明处理器的检查过程。代码清单第69 行将ESP 的初始值设定为0，因此，当第一次进行压栈操作时，假如压入的是一个双字（4 字节）：

```
push ecx
```

因为压栈操作是先减ESP，然后再访问栈，故ESP 的新值是（可以自行用Windows 计算器算一下）

```
0－4＝0xFFFFFFFC
```

这个结果符合上面的限制条件，允许操作。此时，被压入的那个双字，其线性地址为

```
0x00007C00＋0xFFFFFFFC＝0x00007BFC
```

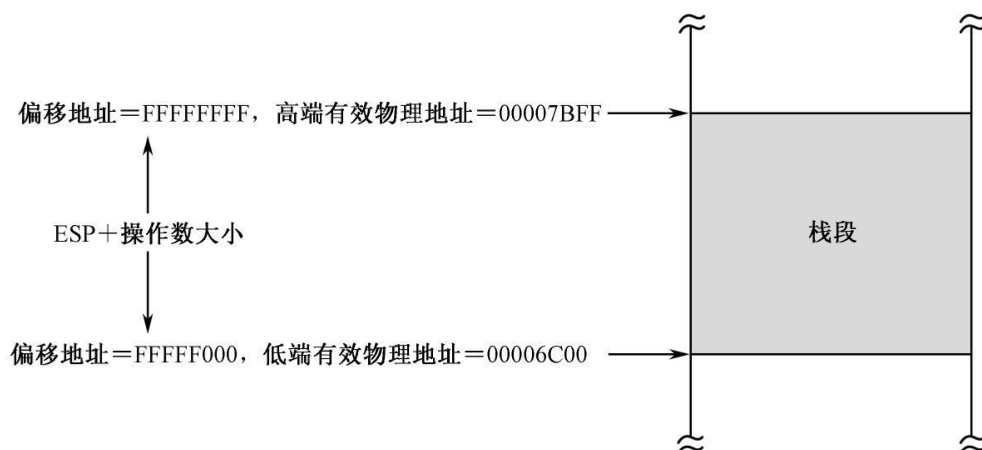


图12-4　对栈段偏移地址的检查

尽管这里讨论的是push 指令，但对于其他隐式操作栈的指令，比如pop、call、ret 等，情况也没有什么不同，也要根据操作数的大小来检查是否违反了段界限的约束，以防止出现访问越界的情况。

12.4.3　数据访问时的保护

这里所说的数据段，特指向上扩展的数据段，有别于栈和向下扩展的数据段。

因为是向上扩展的，所以代码段的检查规则同样适用于数据段。不同之处仅仅在于，对于取指令来说，是否越界取决于指令的长度；而对于数据段来说，则取决于操作数的尺寸。考虑以下指令：

```
mov [0x2000],edx
```

这条指令将访问内存，并将EDX 寄存器的内容写入当前段内偏移量为 0x2000 的双字单元。指令中给出了内存单元的有效地址 EA（0x2000），也给出了操作数的大小（4）。

很好，现在，当处理器访问数据段时，要依据以下规则进行检查：

0≤（EA＋操作数大小－1）≤实际使用的段界限

在任何时候，段界限之外的访问企图都会被阻止，并引发处理器异常中断。

在32 位处理器上，尽管段界限的检查总在进行着，但如果段界限具有最大值，则对任何内存地址的访问都将不会违例。比如本章就定义了一个具有4GB 长的段，段的基地址是0x00000000，段界限是0xFFFFF，粒度为4KB。因此，实际使用的段界限是

0xFFFFF×0x1000＋0xFFF＝0xFFFFFFFF

在这样的段内，访问任何一个内存单元都是允许的，针对段界限的检查都会获得通过。

在32 位模式下，处理器使用32 位的段基地址加上32 位的偏移量，共同形成32 位的物理地址来访问内存。段基地址由段描述符指定，而偏移量由指令直接或者间接给出。很显然，在段最大的时候，可以自由访问4GB 空间内的任何一个单元。

代码清单12-1 第71～74 行，从物理地址0x000B8000 开始写入16 字节的内容，用于演示4GB 内存地址空间的访问。段寄存器ES 当前正指向0 到4GB 的内存空间，其描述符高速缓存器中的基地址是0x00000000，加上指令中提供的32 位偏移量，所访问的地方正是显示缓冲区（显存）所在的区域。这其中的道理很简单，首先，内存的寻址依赖于段基地址和偏移地址，段基地址是0，所以，可以把任何要访问的物理地址作为偏移量。

这16 字节的内容是8 个字符的ASCII 码，以及它们各自的显示属性（颜色）。如图12-5 所示，和往常一样，双字在内存中的写入依然是低端字节序的，这里再次展示一下，以帮助理解。

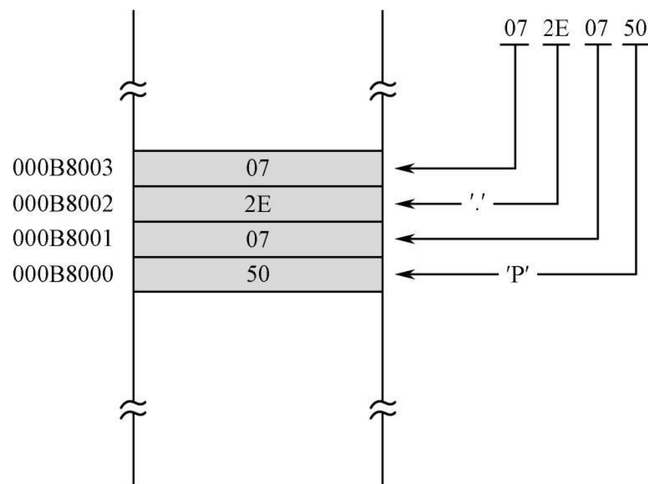


图12-5　以低端字节序向内存中写入双字

要理解32 位模式下的寻址，以及数据访问时的保护机制，这是一个很好的例子。

检测点12.2

当前栈段描述符的B 位是1，基地址为0x00700000，界限值为0xFFFFE。那么，在32 位模式下，该栈段的有效地址范围是0x00700000～（　　　）。当ESP 的内容为0xFFFFF002 时，还能压入一个双字吗？为什么？

12.5　使用别名访问代码段对字符排序

接下来要做的事情是对一串散乱的字符进行排序。坦白地说，排序是假，主要目的是演示如何在保护模式下使用别名段。

字符串位于代码清单12-1 的第105 行，用标号string 声明，并初始化为以下字符：

```
s0ke4or92xap3fv8giuzjcy5l1m7hd6bnqtw.
```

这串字符是主引导程序的一部分，在进入保护模式时，它就位于32位代码段中。代码段是用来执行的，能不能读出，取决于其描述符的类别字段。但是无论如何，它都不允许写入。

这可就难办了。我们想就地把这串字符按ASCII 码从小到大排列，涉及原地写入数据的操作。好在前面已经建立了代码段的别名描述符，而且用段寄存器DS 指向它。参见代码清单12-1第59、60 行。

冒泡排序是比较容易理解的排序算法，但却并不是效率最高的，因此，速度自然也就很慢。如果字符串的长度（字符的数量）是n 个，而且要从小到大排序，那么，可以将它们从头至尾两两比较，需要比较n-1次。但是，不要高兴太早，这一次遍历只会使最大的那个字符慢慢地、像气泡一样移动到最右边。

所以，你需要多次进行这样的遍历才能完成所有字符的排序，每一次遍历都会使一个字符冒泡到正确的位置。可以计算，共需要n-1 次这样的遍历。有关冒泡排序算法的更多信息，请参考其他资料。

可见，这需要两个循环，一个外循环，用于控制遍历次数；一个内循环，用于控制每次遍历时的比较次数。在32 位模式下，loop 指令所用的计数器不是CX，而是ECX。两个循环需要共用ECX，这需要点技巧，那就是利用栈：

```
        mov ecx,n-1           ;控制遍历次数，内、外循环都用它
external:
        xor ebx,ebx           ;清零，从字符串开头处比较
        push ecx

internal:
        …                     ;对字符串两两比较
        inc ebx
        loop internal

        pop ecx
        loop external
```

我相信这段框架性的代码还是很好理解的。外循环总共执行n-1 次。每执行一次外循环，内循环就会将一个数排到正确的位置，从而使下一次内循环少一次两两比对（少执行一次）。也就是说，ECX 寄存器的当前值总是内循环的次数，这就是为什么内循环的loop 指令要使用外循环的ECX 值。

代码清单12-1 第77 行，用后面的标号pdgt 减去声明字符串的标号string，就是字符串的长度，再减去一，就是控制循环的次数。

第79 行，将循环次数压栈，因为内循环会改变ECX 的内容。

第80 行，清零BX 寄存器。该寄存器在每次内部循环之前清零，用于从字符串的开始处进行比对。之所以没有使用EBX，是因为要让你知道，32 位代码中也可以使用16 位的寄存器来寻址。注意，我们知道，在32 位模式下，如果指令的操作数是16 位的，要加前缀0x66。相似地，在32 位模式下，如果要在指令中使用16 位的有效地址，那么，必须为该指令添加前缀0x67。因此，当指令

```
    mov eax,[bx]
```

用bits 32 编译后，会有指令前缀0x67；在32 位模式下执行时，处理器会用数据段描述符中给出的32 位数据段基地址，加上BX 寄存器的16 位偏移量，形成32 位线性地址。

实际进行字符比对的代码是第81～91 行。首先一次性读取两个字符到AX 寄存器中。当前的数据段是由段寄存器DS 指向的，其描述符给出

的基地址为0x00007C00，字符串的首地址就是标号string 的汇编地址，寄存器BX 用来指定字符串内的偏移量。

接着，对寄存器AH 和AL 的内容进行比较。如图12-6 所示，AL 中存放的是前一个字符，AH 中存放的是后一个字符。如果前一个字符较大，则交换AH 和AL 的内容，然后重新写回原来的字单元。然后，将BX 寄存器的内容加一，以指向下一个字符。

xchg 是交换指令，用于交换两个操作数的内容，源操作数和目的操作数都可以是8/16/32 位的寄存器，或者指向8/16/32 位实际操作数的内存单元地址，但不允许两者同时为内存地址。其格式为

```
xchg r/m8,r8
xchg r/m16,r16
xchg r/m32,r32
xchg r8,m8
xchg r16,m16
```

```
xchg r32,m32
```

举个例子：

```
mov ecx,0xf000f000
mov edx,0xabcdef00
xchg ecx,edx
```

以上指令执行后，寄存器ECX 中的内容为0xABCDEF00，EDX 中的内容为0xF000F000。

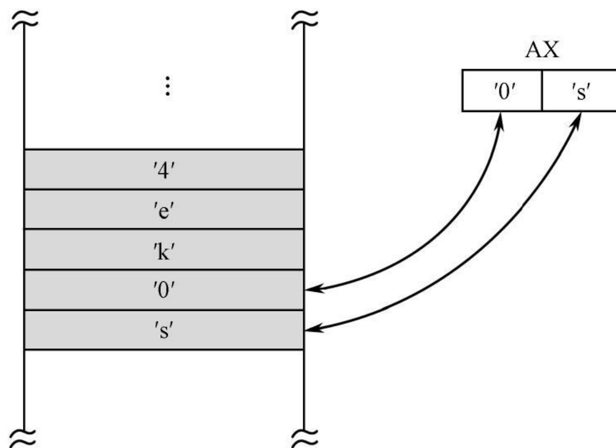


图12-6　通过AX 寄存器比对和排序相邻字符

第93～100 行用于显示最终的排序结果，同样使用了循环，循环次数就是字符串的长度。和排序的时候不同，现在终于使用EBX 了，这将提供32 位的偏移地址。

第96 行，向寄存器AH 传送的是字符的显示属性（颜色），0x07 表示黑底白字，我们已经无数次重复说过了。

第98 行是向显存中传送字符及其显示属性：

```
mov [es:0xb80a0+ebx*2],ax
```

段寄存器ES 是在刚进入保护模式时设置的，它指向0～4GB 内存的段。0xb80a0 等于0xb8000 加上十进制数160（0xa0）。在显存中，偏移量为160 的地方对应着屏幕第2 行第1 列。32 位处理器提供了强大的寻址方式，可以在基址寄存器的基础上使用比例因子，这里是将EBX 寄存器的内容乘以2。当EBX 的内容为0、1、2、3、…时，计算出来的有效地址分别是0xb80a0、0xb80a2、0xb80a4、0xb80a6、…，后面的以此类推，很容易看到使用比例因子的好处。注意，该表达式的值是在本指令执行时，由处理器来计算的。

最后，在完成了所有的工作之后，第102 行，hlt 指令使处理器处于停机状态。

12.6　程序的编译和运行

本章代码清单12-1 所对应的源程序文件是c12_mbr.asm， 用Nasmide 工具将它打开并编译，生成二进制文件c12_mbr.bin 并写入虚拟硬盘的主引导扇区。

然后，启动虚拟机LEARN-ASM，观察运行结果。正常情况下，屏幕显示如图12-7 所示。

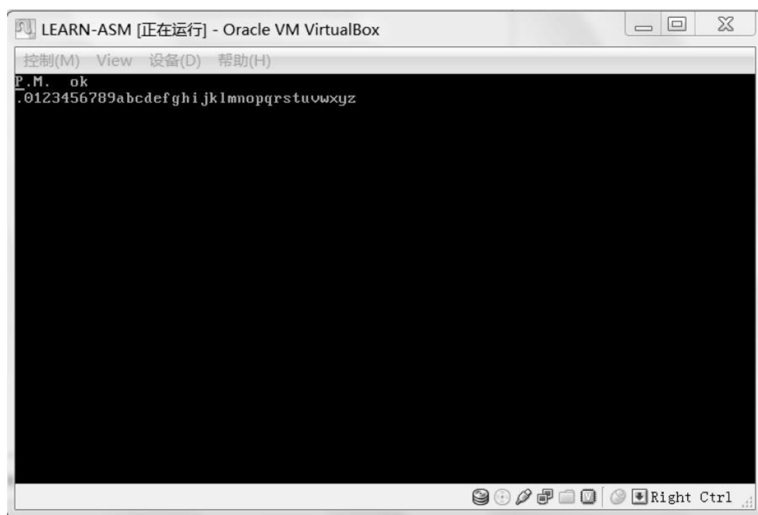


图12-7　本章程序运行结果

本章习题

1．修改本章代码清单，使之可以检测1MB 以上的内存空间（从地址0x00100000 开始，不考虑高速缓存的影响）。要求：对内存的读写按双字的长度进行，并在检测的同时显示已检测的内存数量。建议对每个双字单元用两个花码0x55AA55AA 和0xAA55AA55 进行检测。

第13章　程序的动态加载和执行

像我一样，很多人在了解了保护模式的基本工作原理之后，会产生一个疑问。那就是，所有的段在使用之前，都必须以描述符的形式在描述符表中进行定义，那么，像操作系统这样的软件，又怎么能够加载和执行其他各种用户程序呢？毕竟，你并不知道这些程序都定义了哪些段，每个段是什么类型，有多长。

未必所有人都会产生这样的疑惑，但我确实算一个，可能我还不够聪明。事实上，这仅仅是一层窗户纸，一旦捅破了，才发现原来竟是那么简单。从某种意义上来说，保护模式的工作机制对用户程序的加载和执行非但没有增加困难，反而带来了很大的便利。

一套能够充分说明问题的例子需要很大的代码量，也许把本书的汉字都去掉，全部换成代码也不够。不过，只要能说明问题，也不一定非得完善周全、面面俱到。因此，本章中用于加载和处理用户程序的做法，不一定，甚至根本就不是操作系统采用的方法。这一点，务必明了。

计算机硬件之上是软件。软件分两个层次，一是操作系统，二是应用（用户）程序。通常，用户程序只关心问题的解，就是采用各种算法来解决实际问题。至于软件是怎么加载到内存的，怎么定位的，不是它所操心的事。但是，它有义务提供一些必要的信息，来帮助操作系统将自己加载到内存中。

相反，操作系统则必须考虑采用什么方法来加载用户程序，并在适当的时候将处理器的执行流转移到用户代码中去。同时，为了减轻用户程序的工作量，操作系统还应当管理硬件，并提供大量的例程供用户程序使用。比如，显示一个字符串，就不要让用户自己来写代码了，直接调用操作系统的代码即可。但操作系统和用户程序应当协商一种机制，让用户程序能够在使用这些例程时，不必考虑和关心它们的位置。

本章提供了一个小小的“操作系统”，因为当不起这么大的名称，所以叫“内核”或者“核心”。即使是这样，它依然当不起，因为它实在是太简单了。不过，也没有办法，就这么凑合着叫吧。

内核不能放到主引导扇区里，毕竟它都很大。所以，计算机首先从主引导程序开始执行，主引导程序负责加载内核，并转交控制权。然后，内核负责加载用户程序，并提供各种例程给用户程序调用。提供给用户程序调用的例程也叫应用程序接口（Application Programming Interface，API），本章用简单的方法来允许用户程序使用API 工作。

本章学习目标：

1．了解保护模式是为操作系统提供的技术，并没有给普通应用程序的编程带来负担（这从本章的程序实例中就可以看出来）。

2．学习操作系统在保护模式下加载和重定位应用程序的一般原理，学习简单的内存动态分配，了解应用程序接口API 的简单原理，学习字符串的比较算法。

3．学习若干x86 处理器的新指令，包括bswap、cpuid、cmovcc、sgdt、movzx、movsx、cmpsb、cmpsw、cmpsd 和xlat 等。

13.1　本章代码清单

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：13-1（主引导扇区程序），源程序文件：c13_mbr.asm

本章代码清单：13-2（微型内核），源程序文件：c13_core.asm

本章代码清单：13-3（被加载的用户程序），源程序文件：c13.asm

13.2　内核的结构、功能和加载

13.2.1　内核的结构

内核分为四个部分，分别是初始化代码、内核代码段、内核数据段和公共例程段，主引导程序也是初始化代码的组成部分。

初始化代码用于从BIOS 那里接管处理器和计算机硬件的控制权，安装最基本的段描述符，初始化最初的执行环境。然后，从硬盘上读取和加载内核的剩余部分，创建组成内核的各个内存段。初始化代码大部分位于代码清单13-1 中。

内核的代码和数据位于代码清单13-2 中。如图13-1 所示，内核代码段是在第385 行定义的，用于分配内存，读取和加载用户程序，控制用户程序的执行。

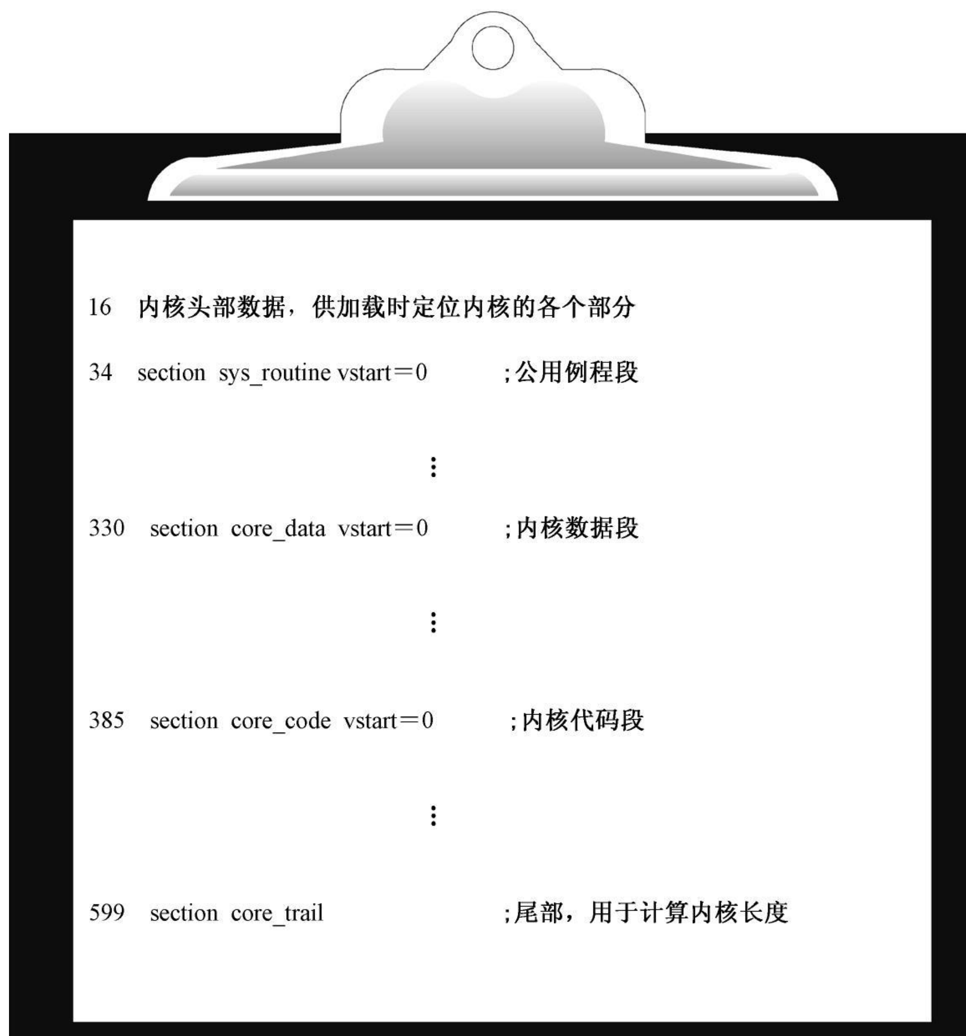


图13-1　内核程序的各个组成部分

内核数据段是在第330 行定义的，提供了一段可读写的内存空间，供内核自己使用。

公共例程段是在第34 行定义的，用于提供各种用途和功能的子过程以简化代码的编写。这些例程既可以用于内核，也供用户程序调用。

除了以上的内容之外，内核文件还包括一个头部，记录了各个段的汇编位置，这些统计数据用于告诉初始化代码如何加载内核。

回到代码清单13-2 的开头。

从第7 行开始，一直到第12 行，用于声明常数。很明显，这是一些内存段的选择子，它们对应的描述符会在内核初始化的时候创建。这些段是内核的段，供内核代码使用，对内核代码是透明的，内核代码“知道”每个段选择子的具体数值，就象你知道自己办公室里有哪些人，可以直

接喊他的名字让他做某件事一样。但是，段选择子的具体数值是和它们在GDT 中的位置相关的。为了不至于在往后因为调整段的位置而修改程序代码，将它们声明成常数是最好的。我们知道，伪指令equ 仅仅是允许我们用符号代替具体的数值，但声明的数值并不占用空间。

内核文件的真正开始部分是头部，偏移量为0x00 的地方是一个双字，可以通过标号core_length 引用，记录了整个内核文件的大小，以字节为单位；偏移量为0x04 的地方是公用例程段的起始汇编地址，是一个双字，可以通过标号sys_routine_seg 引用；偏移量为0x08 的地方是核心数据段的起始汇编地址，也是一个双字，可以通过标号core_data_seg 引用；偏移量为0x0C 的地方是核心代码段的起始汇编地址，双字大小，可以通过标号core_code_seg 引用；从偏移量为0x10 开始的地方用于指示内核入口点，可以通过标号core_entry 引用，在主引导程序加载了内核之后，从这里把处理器的控制权交给内核代码。注意，不要忘了这个表达式，我们以前学过的，它用来得到段的起始汇编地址：

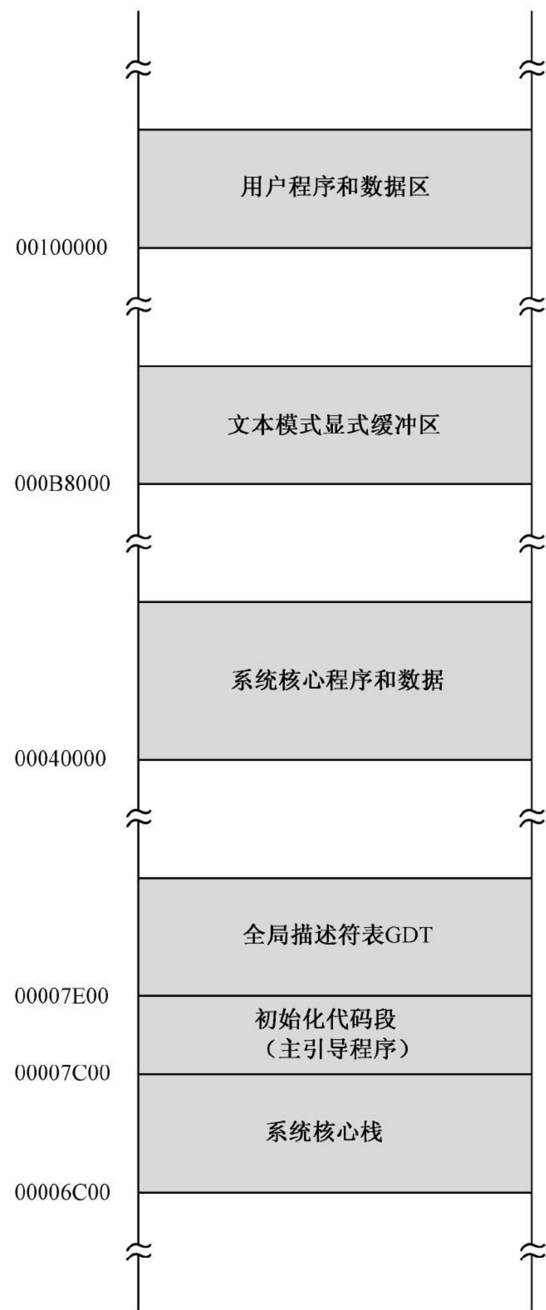


图13-2　本章内存布局示意图

```
section.<段名称>.start
```

入口点共有6 字节，低地址部分是一个双字，指示段内偏移，将来会传送到指令指针寄存器EIP，它来自一个标号start，位于第531 行；高地址部分是一个字，指定一个内存代码段的选择子。在这里，填充的是刚刚在第7 行声明过的常数core_code_seg_sel，在数值上等于0x38。

13.2.2　内核的加载

现在来看代码清单13-1，也就是主引导程序。

第6 行和第7 行声明了两个常数，分别是内核程序在硬盘上的位置，以及它将要被加载的物理内存地址。声明常数的好处你也知道，将来改起来方便。

接下来，从第9 行开始，一直到第55 行，是为进入保护模式做准备。如图13-2 所示，因为主引导程序的加载位置是物理地址0x00007C00，所以，从这个位置往上是512 字节的初始化代码段，从这个位置往下是4KB 的内核栈。

全局描述符表（GDT）是不可或缺的，和从前一样，我们将它定义在从物理地址0x00007E00开始的地方，紧挨着初始化代码段。GDT 可大可小，最大能达到64KB，所以，它的空间一定要留够。

和GDT 一样，内核程序的大小也是不定的，但可以规定它的起始位置。在这里，我们决定将它加载到从物理内存地址0x00040000 开始的地方。从这个地方往上，一直到0x0009FFFF，都是它的地盘，取决于它到底有多大，想用多少就用多少。从0x000A0000 往上，是ROM BIOS，硬件专有的。

显示器是窥视程序工作的窗口，显示功能自然少不了。因此，从0x000B8000 往上的32KB，是文本模式的显示缓冲区。

最后，从1MB 开始的大量空间是留给用户程序用的，具体数量取决于你到底安装了多少物理内存。对于本章来说，程序都很小，功能都很简单，用不了多少内存空间，都才几KB、几十KB；但是，你平时所用的Windows、Linux 和MacOS，以及运行于其上的程序，都是VIP、大客户，动辄几MB、几百MB。

在进入保护模式之前，初始化程序（主引导程序）已经在全局描述符表（GDT）中安装了几个必要的描述符。如图13-3 所示，第一个是用

于访问0~4GB 内存的数据段，它很重要，内核只有在具备了访问全部4GB 内存空间的能力时，才能随心所欲地做任何事情。

第二个是初始化代码段，也就是主引导程序所在的段。进入保护模式后，要继续执行主引导程序的后半部分代码，必须按处理器的要求，为它创建描述符。

最后两个分别是初始的栈段和显示缓冲区的描述符。这里定义的栈在初始化过程中就要使用，而在进入内核之后，它又是内核的栈。

创建这些描述符的代码位于代码清单13-1 的第19～40 行，这几个描述符都和上一章差不多，而且用于创建它们的代码也基本相同，不再逐个讲解。

表内偏移量		描述符索引
+20	文本模式显存（000B8000～000BFFFF）	0x20
+18	初始栈段（00006C00～00007C00）	0x18
+10	初始代码段（00007C00～00007DFF）	0x10
+08	0～4GB数据段（00000000～FFFFFFFF）	0x08
+00	空描述符	0x00

图13-3　进入保护模式前创建的描述符

下面开始加载内核。

首先是初始化各个段寄存器以访问相应的内存段。第59、60 行，使DS 指向全部4GB 的内存空间；第62～64 行，使SS 指向初始的栈空间，并初始化栈指针寄存器ESP 的内容为0。第一个数据压入时，因为栈的操作是先减ESP 的值，再保存数据，所以，如果是压入一个字，ESP 的内容为0xFFFFFFFE；如果压入的是双字，ESP 的内容为0xFFFFFFFC。

接下来是从硬盘把内核程序读入内存，第67～69 行，它在硬盘上的起始逻辑扇区号和物理内存地址已经由两个常数给出，现分别将它们传送到EAX 和EDI 寄存器。

初始化代码并不知道内核有多大，所以也就不知道应该读多少个扇区。不过，它可以先读一个扇区，因为那里包含着内核的头部数据，根

据这些数据，就可以知道内核的总扇区数。

和以前一样， 我们把读硬盘扇区的指令归拢到一起， 做成可以反复调用的过程read_hard_disk_0，它位于第138～192 行。基本上，它的工作过程和具体的代码都和从前一样，但略有不同。首先，该过程要求使用EAX 寄存器来传入28 位的逻辑扇区号。我们现在已经可以使用32 位的寄存器了，再也不会因为16 位寄存器太小，无法容纳28 位的逻辑扇区号而发愁。

其次，这里使用EBX 寄存器来传入偏移地址。因为在32 位模式下，可以访问全部4GB 内存，允许使用32 位的偏移地址。这是好事，我们再也不需要为64KB 的段而受折磨了。

最后一个不同之处在于，每次过程返回时，会使EBX 寄存器的值比原来多512。这是有意的，因为在32 位模式下，内存的访问不再受64KB 限制，所以就能够连续访问。这里，每次将EBX 寄存器的内容加上512，目的是指向下一个内存块，我相信这种工作方式会给调用它的主程序带来方便。

接下来是取得内核的长度，并计算它所占用的扇区数。

因为段寄存器DS 是指向4GB 内存段的，其描述符高速缓存中的基地址是0x00000000，故，第75 行，可以直接用EDI 寄存器中的数值作为偏移量来访问内存，最终生成的线性地址在数值上和EDI 寄存器的内容相同。当前指令的功能是取得内核的总长度，因为它就位于内核的偏移0 处。

第75～77 行，将取得的总字节数除以512，就能在EAX 寄存器中得到内核所占用的扇区数。不过，在没能整除的情况下，实际的扇区总数要比EAX 寄存器中的值多一。

但是，我们要的是剩余扇区数，毕竟已经读了一个。为此，第79～81 行，先判断EDX 寄存器中的余数是否为零。取决于EDX 的实际内容，or 指令会影响ZF 标志位。如果EDX 不为零，则EAX 寄存器里实际上就是剩余的扇区数，因为它比实际的扇区数少一。相反，如果EDX 的内容为零，则EAX 中的内容就是总扇区数，还要用dec 指令减一才行。

无论是哪种情况，指令的执行流程都会到达第83 行。这个地方指令是

```
        or eax,eax
```

这条指令的工作是检查EAX 寄存器，看它的内容是否为零。第84 行，如果为零，说明内核就占用了一个扇区（确实够小的，但一般不太可能），于是不再读硬盘，直接转到标号setup 处执行。

第87～93 行，用于从硬盘读取剩余的扇区，用的是loop 指令循环读取，循环的次数在ECX寄存器中。再重复一遍，**32 位模式下的循环指令需要使用ECX 寄存器**，而不是CX。如果没有第83、84 行的条件判断，而且剩余扇区数为0，那么，这里的循环将执行0xFFFFFFFF＋1 次，显然不是我们希望的。

13.2.3　安装内核的段描述符

要使内核工作起来，首要的任务是为它的各个段创建描述符。换句话说，还要为GDT 续添新的描述符。进入保护模式前，我们在代码清单13-1 的第42 行使用指令

```
lgdt [cs: pgdt+0x7c00]
```

来加载全局描述符表寄存器（GDTR），标号pgdt 所指向的内存位置包含了GDT 的基地址和大小。现在，我们的任务是重新从标号pgdt 处取得GDT 的基地址，为其添加描述符，并修改它的大小，然后用lgdt 指令重新加载一遍GDTR 寄存器，使修改生效。

但是，如果忽略了一件事，你可能不会得逞。标号pgdt 所指向的内存区域位于主引导程序内，而我们当前正在保护模式下执行主引导程序。保护模式下的代码段只是用来执行的，是否能读出，取决于其描述符的类别字段，但无论如何它都不能写入。

对代码段实施保护的意思是通过代码段描述符不能修改段中的内容，但不意味着通过其他描述符做不到。想想看，我们拥有一个指向全部4GB 内存空间的描述符，标号pgdt 所指向的内存位置不单单是在主引导程序内，同时也是4GB 内存空间的一部分。

如图13-4 所示，标号pgdt 在数值上等于它距离段首的偏移量，也就是编译阶段的汇编地址。主引导程序的物理起始地址是0x00007C00，故pgdt 在4GB 段内的偏移量是0x00007C00＋pgdt。

这样，为了得到GDT 的基地址，代码清单13-1 第96 行，使用了指令

```
        mov esi,[0x7c00+pgdt+0x02]
```

注意，指令中的表达式是在编译阶段计算的。默认的段寄存器是DS，当这条指令执行时，处理器用DS 描述符高速缓存器中的32 位线性基地址0x00000000 加上用该表达式计算出的偏移量来访问内存。

现在可以创建与内核相关的其他段描述符。首先是公共例程段。如图13-5 所示，内核头部偏移0x04 处的一个双字，就是公共例程段的起始汇编地址。由于内核被加载的物理地址是由EDI 寄存器指向的，所以，第99 行，直接访问4GB 内存段，从该偏移位置取出公共例程段的起始汇编地址。

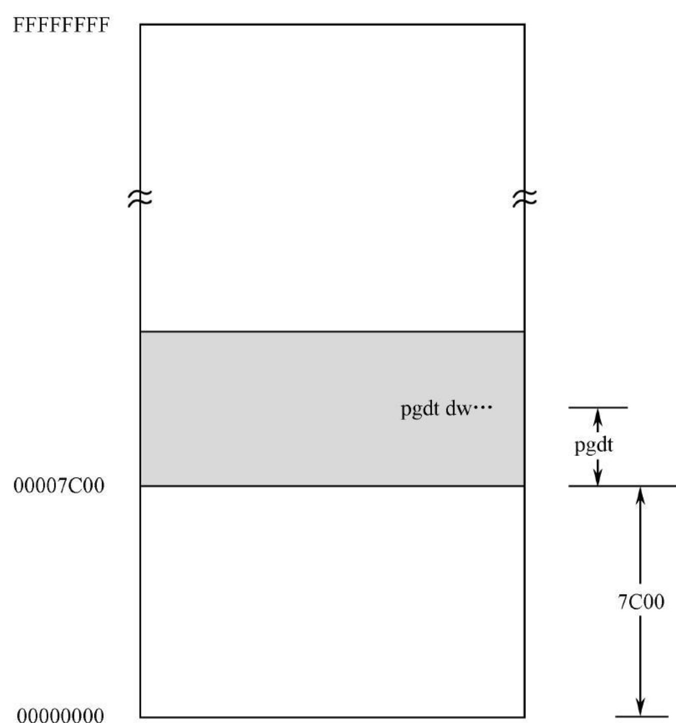


图13-4　通过4GB 数据段访问代码段内的数据

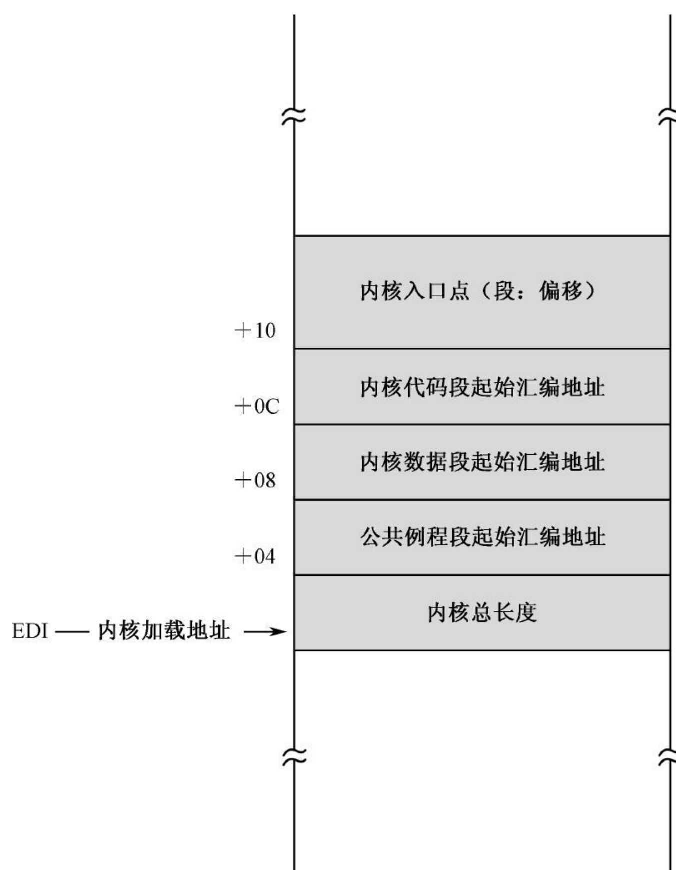


图13-5　内核头部的组成

创建描述符还需要知道段界限。在内核中，各个段有着确定的先后次序，而且是紧挨着的。公共例程段的后面是内核数据段，用内核数据段的起始汇编地址，减去公共例程段的起始汇编地址，再减去一，就是公共例程段的段界限，这就是第100～102 行所做的工作。对于向上扩展的段来说，段界限在数值上等于段的长度减去一，这个必须要清楚。

第103 行，用公共例程段的起始汇编地址，加上内核的加载地址，就是公共例程段的基地址。

在已经知道某个内存段的细节时，写出它的描述符是很容易的。比如，如果已经知道栈的基地址是0x00007C00，粒度是4KB，大小是8KB，那么，它的描述符就可以直接给出：

```
0x00CF96007C00FFFD
```

问题是，这种清楚明白的情形不常见。在百分之九十以上的场合，段的信息只有在程序运行的时候才能确定，它们都是在程序运行时，根

据实际情况得到的随机值。为此，就需要利用指令来以不变应万变，“拼凑”出描述符来。

既然是灵活的方法，还能以不变应万变，就应该定义成过程，以方便在需要的时候随时调用。在这里，我们的方法是使用过程make_gdt_descriptor。

过程make_gdt_descriptor 位于代码清单13-1 的195～217 行，调用该过程需要三个参数，分别是段的线性基地址、段界限和段的属性值。段的基地址用EAX 寄存器传入；段界限用EBX 寄存器传入，但只用其低20 位；段属性用ECX 寄存器传入，各属性位在ECX 寄存器中的分布和它们在描述符高32 位中的时候一样，其他和段属性无关的位都清零。

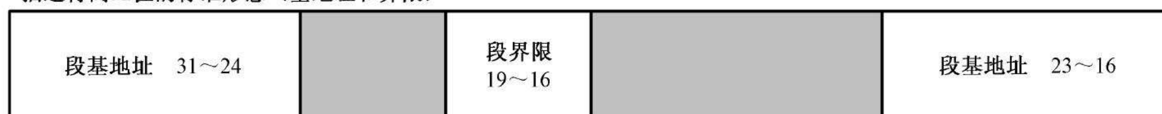
因此，第104 行，将段属性值0x00409800 传送到ECX 寄存器。结合第11 章的图11-4，可以知道，这是一个P＝1、D＝1、G＝0、DPL＝0、S＝1，TYPE＝1000 的（代码）段描述符。第105 行，调用过程创建描述符，下面来看看具体的创建过程。

代码清单13-1 的第201～203 行用于构造描述符的低32 位。首先是将32 位段基地址从EAX 寄存器复制一份给EDX 寄存器，过一会儿构造描述符的高32 位时，还要用到基地址。

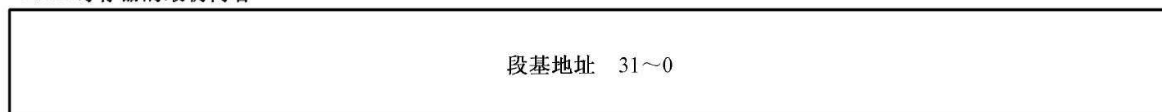
描述符的低32 位中，高16 位是基地址；低16 位是段界限，所以，第202～203 行，将EAX寄存器中的32 位基地址左移16 次，使基地址部分就位。然后，把BX 寄存器中的段界限用or 指令安排就位。这样，描述符的低32 位就构造完毕了。

相比之下，描述符的高32 位构造起来比较麻烦。如图13-6 所示，描述符高32 位的标准形态是有两个基地字段和一个段界限字段。基地址在EDX 寄存器中有备份，执行第205～207 行的指令后，会使基地址部分在两边就位。

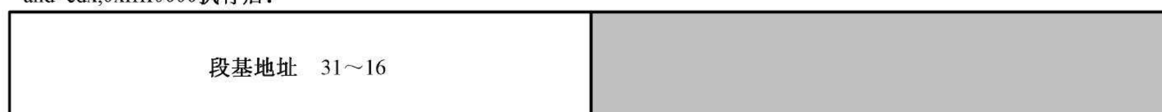
描述符高32位的标准形态（基地址和界限）



EDX 寄存器的最初内容



and edx,0xffff0000执行后：

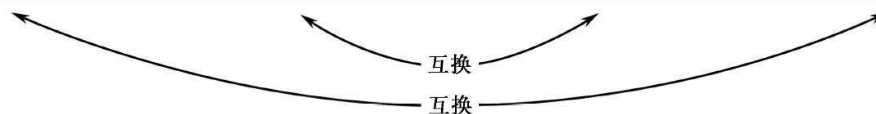


rol edx, 8执行后：



bswap edx执行后：





xor bx, bx执行后



图13-6　描述符高32 位的构造过程

bswap 是字节交换指令（Byte Swap），在标准的32 位处理器上只允许32 位的寄存器操作数，其格式为

```
bswap r32
```

处理器执行该指令时，按如下过程操作（DEST 是指令中的操作数，TEMP 是处理器内的临时寄存器）：

```
TEMP ← DEST
DEST[7:0] ← TEMP[31:24];
DEST[15:8] ← TEMP[23:16];
DEST[23:16] ← TEMP[15:8];
DEST[31:24] ← TEMP[7:0];
```

接下来，要在描述符的高32 位中装配段界限字段。第209、210行，先清除EBX 寄存器的低16 位，然后同EDX 寄存器合并。这里是假设EBX 寄存器的高12 位为全零，所以用了xor bx，bx指令。实际上，安全的做法是使用指令

and ebx,0x000f0000 最后，第212 行，将ECX 寄存器中的段属性与EDX 寄存器中的描述符高32 位合并。至此，我们就在EDX:EAX 中得到了完整的64 位描述符。第214 行，ret 指令将控制返回到调用者。

现在，回到主程序，来看第106、107 行，ESI 寄存器的内容是GDT的基地址，这两条指令访问4GB 的段，定位到GDT，在原先的基础上，再添加一个描述符，就是我们刚刚创建的描述符。

第110～129 行，用于安装内核数据段和内核代码段的描述符，也采用了相同的过程，不再一一讲解。

第131 行，通过4GB 的数据段访问pgdt，修改它的界限值。现在，GDT 中已经有8 个描述符，故其总长度为64 字节。相应地，界限值为63。

第133 行，通过4GB 的数据段访问pgdt，重新加载GDTR，使上面那些对GDT 的修改生效。

至此，内核已经全部加载完毕，图13-7 是内核加载完成之后的GDT布局。

第136 行，通过4GB 的数据段访问内核的头部，用间接远转移指令从给定的入口进入内核执行。观察图13-5，再参考代码清单13-2 就可以明白，在内核头部偏移0x10 处，是6 字节的内核入口点。前面是32 位的段内偏移地址，后面是16 位的段选择子，指向内核代码段。在这里，段选择子直接使用固定的数值不是一个好主意，怕的是往后内核有重大调整时，会改变描述符的次序。在这种情况下，如果别处改了，这里忘了修改，就一定会出现问题。

表内偏移量		描述符索引
+38	核心代码段（位于核心数据段之后）	0x38
+30	核心数据段（位于系统公用例程段之后）	0x30
+28	公用例程段（起始地址为00040000）	0x28
+20	文本模式显存（000B8000～000BFFFF）	0x20
+18	初始栈段（00006C00～00007C00）	0x18
+10	初始代码段（00007C00～00007DFF）	0x10
+08	0～4GB数据段（00000000～FFFFFFFF）	0x08
+00	空描述符	0x00

图13-7　内核加载完成后的GDT 布局

13.3　在内核中执行

现在转到代码清单13-2，这是内核的主体部分。

从主引导程序转移到内核之后，处理器会从第532 行开始执行，因为这里是内核的入口。

第532、533 行，初始化段寄存器DS，使它指向内核数据段。然后，第535、536 行，调用公共例程段内的一个过程来显示字符串。该call 指令属于直接远转移，指令中给出了公共例程段的选择子和段内偏移量。字符串是在第362 行，用标号message_1 声明，并初始化了一段文字，意思是"如果你看到这段信息，那么这意味着我们正在保护模式下运行，内核已经加载，而且显示例程工作得也很完美。"

显示例程put_string 位于公共例程段内，是在第37 行定义的。基本上，它的代码组成和工作原理都和从前一样，但也有不同之处。首先，这里的代码是32 位模式的，字符串的地址由DS:EBX 传入，过程返回时用retf 指令，而不是ret。这意味着，必须以远过程调用的方式使用它。

和往常一样，put_string 在内部调用了另一个过程put_char。注意，第110～113 行，movsd 用于在两个内存区域间传送双字数据（一次传送4 字节）。不管是movsb，还是movsw，抑或是movsd，在16 位模式下，是把由DS:SI 指定的源操作数传送到由ES:DI 指定的目的地。但是，在32 位模式下，源和目的则分别是DS:ESI 和ES:EDI。

再回到539 行，下面的工作是显示处理器品牌信息。

处理器的功能是强劲的，这个没有人怀疑。同时，在处理器内部也隐藏着太多的秘密，除了处理器的型号，还有大量的特性信息，比如高速缓存的数量、是否具备温度和电源管理功能、逻辑处理器的数量、高级可编程中断控制器的类型、线性（物理）地址的宽度、是否具有多媒体扩展和单指令多数据指令等特性。

处理器功能强了是好事，大家都很欢喜。麻烦在于，很多新功能是处理器更新换代的产物，只存在于最新的版本中，旧的处理器没有。比如多媒体扩展指令可以加速多媒体的处理速度，但用了新指令的软件不能运行在旧的处理器上，因为它们不支持。可怕之处在于，没有人知道

自己的软件被终端销售商卖给了谁，更不知道那个谁用的是什么处理器。

因此，你的软件应当准备两套方案，而且，在决定使用哪套方案之前，必须探测和挖掘处理器内部的秘密，好知道该怎么办。Intel 公司显然洞悉了市场上发生的一切，它们给出的方案是使用cpuid 指令。

cpuid 指令（CPU Identification）用于返回处理器的标识和特性信息。EAX 用于指定要返回什么样的信息，也就是功能。有时候，还要用到ECX 寄存器。cpuid 指令执行后，处理器将返回的信息放在EAX、EBX、ECX 或者EDX 中。

cpuid 指令是从80486 处理器的后期版本开始引入的，从此以后，每款处理器都会对可以返回的信息有所扩充。原则上，在使用cpuid 指令前，先要检测处理器是否支持该指令；接着再用cpuid 指令检测是否支持所需要的功能。

如图13-8 所示，在32 位处理器上，原先的标志寄存器FLAGS 也相应地扩充到了32 位，以支持更多的标志。扩充之后的标志寄存器称为EFLAGS 寄存器，它的ID 标志位（位21）如果为“0”，则不支持cpuid 指令；反之，该处理器支持cpuid 指令。80486 处理器已经很久远了，我想没有谁还在使用这样的计算机，况且它已经停产。一般情况下，不需要检测处理器是否支持cpuid 指令。

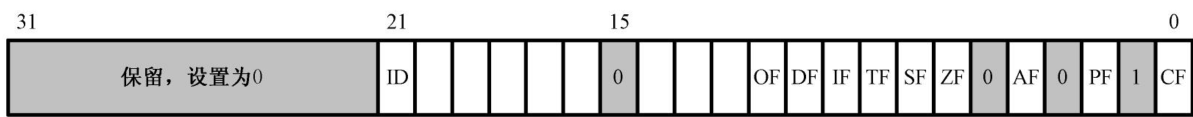


图13-8　扩展到32 位长度的标志寄存器EFLAGS

图13-8 中，灰色的部分是保留位，通常设置为固定的值。EFLAGS 还包括更多的标志位，图中未予显示，仅在以后用到的时候一一介绍。

为了探测处理器最大能够支持的功能号，应该先用0 号功能来执行cpuid 指令：

```
mov eax,0
cpuid
```

处理器执行后，将在EAX 寄存器返回最大可以支持的功能号。同时，还在EBX、ECX 和EDX 中返回处理器供应商的信息。对于Intel 处理

器来说，返回的信息如下：

```
EBX ← 0x756E6547 (对应字符串“Genu”，“G”在BL中，其他类推)
EDX ← 0x49656E69 (对应字符串“ineI”，“i”在DL中，其他类推)
ECX ← 0x6C65746E (对应字符串“ntel”，“n”在CL中，其他类推)
```

组合起来就是“GenuineIntel”。

要返回处理器的品牌信息，需要使用0x80000002～0x80000004 号功能，分三次进行。注意，该功能仅被奔腾4（Pentium 4）之后的处理器支持，所以，正确的做法是先用0 号功能执行cpuid指令，以判断自己的处理器是否支持。代码清单13-2 并没有这样做，因此可视为一个反面典型。

第539～558 行，分别用三种功能号执行cpuid 指令，返回三组字符串，共48 个字符，依次保存在核心数据段中，起始位置是由标号cpu_brand 指定的。第381 行，声明了标号cpu_brand，并初始化了52 字节，足以容纳这些数据。

从处理器返回的数据都是现成的ASCII 码。第560～565 行，先在屏幕上留出空行，再显示处理器品牌信息，然后再留空，以突出要显示的内容。

13.4 用户程序的加载和重定位

好了，现在我们可以开始加载用户程序了。

用户程序加载之前，要先显示一段信息，意思是要加载用户程序了。这是第567、568 行的工作。字符串位于内核数据段中，第367 行声明了标号message_5 并初始化了字符串。

第569 行用于指定用户程序的起始逻辑扇区号。在指令中直接指定数值不是一个好习惯，正确的做法是用伪指令equ 声明成常数，并放到整个程序的起始部分以便修改。

内核的主要任务就是加载和执行用户程序。通常情况下，这样的工作会反复进行。为了方便， 一般要定义成可反复调用的过程。在这里，我们也是这样做的， 过程的名字叫load_relocate_program。该过程位于第387 行，作用是加载和重定位用户程序。从代码清单中可以看出，它是内核代码段的一个内部过程。

13.4.1 用户程序的结构

用户程序必须符合规定的格式，才能被内核识别和加载。通常情况下，流行的操作系统会规定自己的可执行文件格式，一般都比较复杂，这种复杂性和操作系统自身的复杂性是息息相关的。

现在转到代码清单13-3，来看看用户程序的结构。

所有操作系统的可执行文件都包括文件头，这里也不例外。事实上，这也是我们熟悉的、一贯的做法。在文件头内的偏移0 处，是一个双字，指示了用户程序的大小，以字节为单位。

偏移量为0x04 处的双字是头部的长度，以字节为单位。

偏移量为0x08 处的双字是为栈保留的，和早先的做法不同，内核不要求用户程序提供栈空间，而改由内核动态分配，以减轻用户程序编写的负担。当内核分配了栈空间后，会把栈段的选择子填写到这里，用户程序开始执行时，可以从这里取得该选择子以初始化自己的栈；

偏移量为0x0c 处的双字是要求分配的栈大小，即，用户程序编写者建议的栈大小，以4KB为单位。如果是1，就是希望分配4KB 的栈空间；如果是2，就是希望分配8KB 的栈空间，依此类推。

偏移量为0x10 处的双字，是用户程序入口点的32 位偏移地址。

偏移量为0x14 处的双字，是用户程序代码段的起始汇编地址。当内核完成对用户程序的加载和重定位后，将把该段的选择子回填到这里（仅占用低字部分）。这样一来，它和0x10 处的双字一起，共同组成一个6 字节的入口点，内核从这里转移控制到用户程序。

偏移量为0x18 处的双字，是用户程序代码段的长度，以字节为单位。

偏移量为0x1c 处的双字，是用户程序数据段的起始汇编地址，当内核完成用户程序的加载和重定位后，将把该段的选择子回填到这里（仅占用低字部分）。

偏移量为0x20 处的双字，是用户程序数据段的长度，以字节为单位。

除了加载和重定位用户程序外，内核还应当提供一些例程供用户程序调用。操作系统对于普通用户来说，是赏心悦目的界面和快捷直观的操作方式，对程序员来说，则是一个巨大的例程库，节省了时间，减少了工作量，甚至不需要直接访问硬件。

操作系统提供的编程接口就是API，这是一大堆例程（过程），需要的时候直接调用即可。问题在于，它们在操作系统内部，对任何人来说都是不可见的，更别想知道它们的入口地址。但是，call 指令是需要直接或间接提供一个地址的。另一方面，即使你知道它们的地址，调用的时候也有风险，因为操作系统也需要升级换代，这些地址可能改变。当你的程序在新操作系统上工作时，就要出问题。

为了使开发人员能够利用它所提供的API，操作系统至少要公开它们。在早期的系统中，这些API 以中断号的方式公布，因为它们是通过软中断进入的。不过，另一种常见的办法是使用符号名。比如，操作系统提供了一个例程，用于显示光标跟随的字符串，那么，它可以公布一个符号名：

```
PrintString
```

当然，它肯定不会同时公布一个段地址和偏移地址，因为它也不能保证地址不会变化。在操作系统的开发手册中，会列出所有符号名。符号名在高级语言里就是库函数名。

回到代码清单13-3 中来。

内核要求，用户程序必须在头部偏移量为0x28 的地方构造一个表格，并在表格中列出所有要用到的符号名。每个符号名的长度是256 字节，不足部分用0x00 填充，这意味着每个符号名的长度最多可以是256 个字符。在用户程序加载后，内核会分析这个表格，并将每一个符号名替换成相应的内存地址，这就是过程的重定位。为了方便起见，我们把该表格叫做“符号-地址检索表”（Symbol-Address Lookup Table，SALT）。不要上网搜索这个词，也不要查别的资料，这不是一个标准，是我自己随心所欲、特立独行的产物。

第29～36 行声明了三个标号，并分别初始化了三个符号名，每一个256 字节，不足部分是用0 填充的。每个符号名都以“@”开始，这并没有任何特殊意义，仅仅在概念上用于表示“接口”的意思。为了计算需要填充多少个0，它们都使用了相似的表达式，比如：

```
        times 256-($-PrintString) db 0
```

这里，先计算出符号名的实际字符数，即$-PrintString，再用256 减去实际字符数，就得到了伪指令db 的重复次数。

SALT 表可大可小，内核需要知道它在哪里结束。第26 行，用于初始化SALT 表的项数，也就是符号名的数量，它是用表格的总长度除以每个符号名的长度（256）得到的。

事实上，即使是大多数汇编语言，也不需要亲自构造文件头，那是链接器（Linker）的工作。但是，链接器是为流行的操作系统服务的，用于构造他们可以识别的可执行文件格式。我们不想把问题搞得太复杂，就本书的篇幅和宗旨来说，迎合“流行”所要花费的代价实在太大，管中窥豹、点到即止不是很好吗？

13.4.2　计算用户程序占用的扇区数

再次回到代码清单13-2。

用户程序的加载是在例程load_relocate_program 内进行的，该过程需要用ESI 寄存器传入用户程序的起始逻辑扇区号。当过程返回时，在AX 寄存器内包含了指向用户头部段的选择子。

第396、397 行，因为在过程中要用到DS 和ES，故将其原先的内容压栈保存。

为了得到用户程序的大小，需要先预读它的第一个扇区，第399～404 行就在做这件事。首先，使段寄存器DS 指向内核数据段；然后，调用读硬盘的过程read_hard_disk_0 来预读用户程序。进入过程前，EAX 寄存器的内容是用户程序的起始逻辑扇区号；数据的存放地点是内核缓冲区core_buf，它位于内核数据段中，是在第376 行声明和初始化的。在内核中开辟出一段固定的空间，对于分析、加工和中转数据都比较方便。

接下来的工作是计算用户程序到底占用了多少个扇区。用户程序的总大小就在头部内偏移量为0x00 的地方，因此，第407 行直接访问内核缓冲区取得这个双字。

用户程序的大小（总字节数）不一定恰好是512 的整数倍。也就是说，最后一个扇区未必是满的。因此，如果直接除以512，可能会使结果（除法的商）比实际的扇区数少一。通常情况下，需要判断除法的余数，根据余数是否为零，来决定实际的扇区总数，这不可避免地要使用判断和条件转移指令。

在早先的处理器中，转移指令是影响处理器速度的重大因素之一，因为它会使流水线中那些已经预取和译码的指令失效。在较晚的处理器中，普遍采用了分支预测技术，但并不总能保证预测是准的。因此，最好的办法就是尽量不使用转移指令。为了帮助程序员部分地戒掉使用转移指令的欲望，处理器引入了条件传送指令cmovcc。

cmovcc 指令是从P6 处理器族开始引入的，因此并非所有处理器都支持它。如果你想知道确切的结果，可以先以1 号功能执行cpuid 指令：

```
mov eax,1
cpuid
```

当处理器执行这两条指令后，会在EBX、ECX 和EDX 寄存器返回丰富的信息，以指示各种详尽的处理器特性。此时，检查EDX 寄存器的第16 位（bit 15），当它是“1”时，表明处理器支持cmovcc 指令。

条件转移指令和传送指令相结合的产物，既有条件转移指令的多样性，又执行的是传送操作。但是，和mov 指令不同的是，它的目的操作数只允许是16 位或者32 位通用寄存器，源操作数只能是相同宽度的通用寄存器和内存单元，以下是几个常用的例子：

```
cmovz ax,cx                 ;为零则传送
cmovnz eax,[0x2000]         ;不为零则传送
cmove ebx,ecx               ;相等则传送
cmovng cx,[0x1000]          ;不大于则传送
cmovl edx,ecx               ;小于则传送
```

条件传送指令是很多的。在第6 章的表6-1 中，列举了所有的条件转移指令。完整的cmovcc指令列表，可以在表6-1 的基础上，将那些指令的首字母“j”换成“cmov”即可。

cmovcc 指令不影响EFLAGS 寄存器中的任何标志位。相反地，它的执行过程要依赖于这些标志，就像条件转移指令一样。

言归正传，为了不使用条件转移指令而又能算出用户程序实际占用的扇区数，需要一点技巧。考察一下，你发会现，所有能被512 整除的数，其最低端的9 个比特都是“0”。比如：

```
0x200（对应的十进制数为 512）  -> 0000 0010 0000 0000B
0x400（对应的十进制数为 1024） -> 0000 0100 0000 0000B
0x600（对应的十进制数为 1536） -> 0000 0110 0000 0000B
0x800（对应的十进制数为 2048） -> 0000 1000 0000 0000B
0xE00（对应的十进制数为 3584） -> 0000 1110 0000 0000B
```

很好，第408 行，将用户程序的总大小从EAX 寄存器传送到EBX 寄存器，等于是做个备份，因为后面还要用到；第409、410 行，先用and指令将其最低的9 个比特清零，等于是去掉那些不足512 的零头，然后，再将其加上512，等于是将那些零头凑整。

但是，若人家原本就是512 的整数倍，你这么做无疑是多加了一个扇区。因此，第411、412 行，先测试EAX 寄存器的最低9 个比特，如果测试的结果是它们不全为零，则采用凑整的结果；如果为全零，则cmovcc 指令什么也不做，依然采用用户程序原本的长度值。

13.4.3　简单的动态内存分配

下面的工作是把用户程序从硬盘上读到内存中。我们以前的做法是指定一个区域，比如物理地址0x100000，然后把程序加载到那里。如果要加载的程序很多，这就会成为一种需要仔细规划的工作，每个程序加载到哪里，都需要一一指定。

在流行的操作系统里，内存管理是一项重要而又严肃的工作，不用说也相当复杂。它要记住所有可以分配的内存，将它们分成块。这样，当要求分配内存时，内存管理程序将查找并分配那些大小相符的空闲块；当占用这些块的用户终止执行后，还要负责回收它们，以便再用于分配；当内存空间紧张，找不到空闲块，或者空闲块的大小不能满足需求时，内存管理程序还要负责查找那些很少被访问的块，将其中的数据移到硬盘上，腾出空间来满足当前的需求。下次当这些块再次被用到时，再用同样的办法从硬盘调回内存。

讲了这么多，你可能以为我们现在就要写一个内存管理程序。不，不会的，这不太现实。就我们目前的需求来说，只需要一个简单的内存分配程序就可以了，这就是allocate_memory 例程。

allocate_memory 例程位于代码清单13-2 的公共例程段中，它仅仅需要通过ECX 寄存器传入希望分配的字节数。当过程返回时，ECX 寄存器包含了所分配内存的起始物理地址。

allocate_memory 的内存分配策略非常简单。请看代码清单13-2 的第335 行，在内核数据段中声明了标号ram_alloc，并初始化为一个双字0x00100000，这就是可用于分配的初始内存地址。很显然，这个位置正好在1MB 之外。每次请求分配内存时，allocate_memory 过程仅简单地返回该内存单元的值，作为所分配内存的起始地址。同时，将这个值加上所分配的长度，作为下次分配的起始地址写回该内存单元。

因此，在进行了必要的现场压栈保护之后，第239～247 行，先使段寄存器DS 指向内核数据段以访问标号ram_alloc 所指向的内存单元；然后，计算下次可用于分配的起始内存地址并存放到EAX 寄存器中；最后，在ECX 中得到本次分配到的起始内存地址，这个值将返回给调用者。当然，在这个过程中没有检测是否超越了实际拥有的物理内存。我们的程序都非常小，现在哪台计算机没有几十兆、几百兆甚至几个吉的内存呢？

原则上，将EAX 寄存器中的值写回ram_alloc 所指向的双字单元即可。不过，32 位的计算机系统建议内存地址最好是4 字节对齐的，这样

做的好处是访问速度最快。为此，在将EAX 寄存器的值写回内存之前，最好使之成为可被4 整除的值，这种数值的特点是最低两个比特为“0”。

第249～254 行，先将EAX 寄存器的内容传送到EBX 进行备份；接着，强制EBX 中的地址对齐在下一个4 字节边界，对齐之后的值肯定会比原先大；然后，看一看原始分配的起始地址（在EAX 寄存器中）是否是4 字节对齐的，如果不是，就采用对齐之后的值；如果原本就是4 字节对齐的，那么，依然采用原值；最后，将这个值写回到原内存单元中，作为下次内存分配的起始地址。

过程allocate_memory 是用retf 指令返回的。因此，它只能通过远过程调用来进入。

13.4.4　段的重定位和描述符的创建

接着回到load_relocate_program 过程。

在13.4.2 节里，我们算出了用户程序的总长度，而且已经被调整为可以被512 整除的数。第414、415 行，用这个数值去调用allocate_memory 过程分配内存。分配到手的内存块，起始地址在ECX 寄存器中。

第416 行，将ECX 寄存器的内容传送到EBX，其动机是作为起始地址从硬盘上加载整个用户程序。

第417 行，将该首地址压栈保存，其目的是用于在后面访问用户程序头部。

第418～420 行，用户程序的总长度除以512，得到它所占用的扇区总数。

第421 行，将扇区数传送到ECX 寄存器，用于控制后面的循环次数。该循环是用来加载整个用户程序的。

第423、424 行，使段寄存器DS 指向4GB 的内存段，这样就可以加载用户程序了。

第428～430 行，循环读取硬盘以加载用户程序。读取的次数由ECX 控制；加载之前，其首地址已经位于EBX 寄存器。起始逻辑扇区号原本是通过ESI 寄存器传入的，循环开始之前已经传送到EAX 寄存器（第426 行）。

既然用户程序已经全部读入内存，现在的任务就是根据它的头部信息来创建段描述符。

第433 行，从栈中弹出用户程序首地址到EDI 寄存器，它是在前面第417 行压入的，该地址也是用户程序头部的起始地址。

第434～438 行，读用户程序头部信息，根据这些信息创建头部段描述符。在主引导程序里，有一个创建描述符的例程，在内核中，也编写了一个同样的例程make_seg_descriptor，甚至它们所用的指令都一模一样。它属于公共例程段，是在第308 行定义的。

该过程要求用EAX 寄存器传入段的基地址，这是第434 行的工作。段界限由EBX 寄存器传入，第435、436 行访问4GB 内存段，从用户程序头部偏移0x04 处取出段长度，减一后形成段界限。第437 行用于给出头部段的属性值。

从过程返回时，EDX：EAX 中包含了64 位的段描述符。紧接着，第439 行调用公共例程段内的另一个过程set_up_gdt_descriptor，把该描述符安装到GDT 中。

set_up_gdt_descriptor 也属于公共例程段，是在第263 行定义的，它需要通过EDX:EAX 传入描述符作为唯一的参数。该过程返回时，CX 寄存器中包含了那个描述符的选择子。

要在GDT 内安装描述符，必须知道它的物理地址和大小。而要知道这些信息，可以使用指令sgdt（Store Global Descriptor Table Register），它用于将GDTR 寄存器的基地址和边界信息保存到指定的内存位置。sgdt 指令的格式为

```
sgdt m
```

其中，m 是一个6 字节内存区域的首地址。该指令不影响任何标志位。

第332、333 行，在内核数据段中，声明了标号pgdt，并初始化了6 字节，供sgdt 指令使用。低2 字节用于保存GDT 的界限（大小）；高4 字节用于保存GDT 的32 位物理地址。

回到例程set_up_gdt_descriptor 中。第270～276 行，在压栈保存了DS 和ES 的原始内容后，使DS 指向内核数据段。紧接着，使用sgdt 指令取得GDT 的基地址和大小。

第278、279 行，使段寄存器ES 指向4GB 内存段以操作全局描述符表（GDT）。

下面的工作是计算描述符的安装地址。这个地址可以这样计算：先得到描述符表的界限值，将它加一，得到描述符表的总字节数，这实际上也是新描述符在GDT 内的偏移量。然后，用GDT 的线性地址加上这个偏移量，就是用于安装新描述符的线性地址。

第281 行，先访问内核数据段，取得GDT 的界限值。注意，这里出现了一个新指令movzx，其作用是带零扩展的传送（Move with Zero-Extend），指令格式为

```
movzx r16,r/m8
movzx r32,r/m8
movzx r32,r/m16
```

也就是说，movzx 指令的目的操作数只能是16 位或者32 位的通用寄存器，源操作数只能是8位或者16 位的通用寄存器，或者指向一个8 位或16 位内存单元的地址。而且，很有意思的是，目的操作数和源操作数的大小是不同的。这里有几个例子：

```
movzx cx,al
movzx eax,byte [0x2000]
movzx ecx,bx
```

对于上面的第一个例子，如果指令执行前，AL 寄存器的内容是0xC0，那么，指令执行后，CX 寄存器的内容为0x00C0；对于第二个例子，处理器访问段寄存器DS 所指向的段，从偏移地址0x2000 处取得一字节，左边添加24 个“0”，使之扩展到32 位，然后传送到EAX 寄存器；对于第三个例子，如果指令执行前，BX 寄存器的内容为0x55AA，那么，指令执行后，ECX 寄存器的内容为0x000055AA。

另一个非常有用的指令是movsx，意思是带符号扩展的传送（Move with Sign-Extension），指令格式为

```
movsx r16,r/m8
movsx r32,r/m8
movsx r32,r/m16
```

和movzx 不同，movsx 在执行扩展时，用于扩展的比特取自源操作数的符号位。比如

```
mov al,0x08
movsx cx,al         ;CX=0x0008，因为 AL 的最高位是“0”

mov al,0xf5
movsx ecx,al        ;ECX=0xFFFFFFF5，因为 AL 的最高位是“1”
```

GDT 的界限是16 位的，允许64KB 的大小，即8192 个描述符，似乎不需要使用32 位的寄存器EBX。事实上，还是需要的，因为后面要用它来计算新描述符的32 位线性地址，加法指令add要求的是两个32 位操作数。

第282 行，将GDT 的界限值加1，就是GDT 的总字节数，也是新描述符在GDT 内的偏移量。不过，很奇怪的是，我们用的是指令

```
inc bx
```

而不是

```
inc ebx
```

这是为什么呢？

这是有道理的。就一般的情况来说，在这里用这两条指令的哪一条，都没有问题。但是，如果这是启动计算机以来，第一次在GDT 中安装描述符，可能就会产生问题。在初始状态下，也就是计算机启动之后，这时还没有使用GDT，GDTR 寄存器中的基地址为0x00000000，界限为0xFFFF。

当GDTR 寄存器的界限部分是0xFFFF 时，表明GDT 中还没有描述符。因此，将此值加1，结果是0x10000，由于该寄存器的界限部分只有16 位，所以只能容纳16 位的结果，即0x0000，这就是第一个描述符在表内的偏移量。

同样的道理，因为EBX 寄存器中的内容是GDT 的界限值0x0000FFFF，如果执行的是指令

```
inc ebx
```

那么，EBX 寄存器中的内容将是0x00010000，以它作为第一个描述符的偏移量显然是不对的。相反，如果执行的是指令是

```
inc bx
```

那么，因为BX 寄存器只有16 位，故，结果为0x0000，进位被丢弃（决不会影响EBX 寄存器的高16 位）。此指令执行后，EBX 寄存器的内容是0x00000000。

第283 行，用计算出来的偏移量加上GDT 的基地址，结果就是新描述符的线性地址。事实上，这三行或许可以按以下方法来简单处理，就没那么啰嗦了：

```
xor ebx,ebx
mov bx,[pgdt]               ;GDT 界限
inc bx                       ;GDT 总字节数，也是下一个描述符偏移
add ebx,[pgdt+2]             ;下一个描述符的线性地址
```

但是，少用一条指令似乎更好，谁知道呢！

既然已经知道新描述符应该安装在哪里，第285、286 行，访问段寄存器ES 所指向的4GB 内存段，将EDX:EAX 中的64 位描述符写入由EDI 寄存器所指向的偏移处。

第288～290 行，访问内核数据段，将GDT 的界限值加上8，然后用lgdt 指令重新加载GDTR，使新的描述符生效。GDTR 寄存器中的界限值总是单数（8 的整数倍减1），包括它的初始值0xFFFF。所以，每次只要加上新描述符的实际大小就能得到正确的界限值。

最后，第292～297 行，根据GDT 的新界限值，来生成相应的段选择子。具体的算法是，取得GDT 的当前界限值，除以8，余数丢弃。描述符的索引是从0 开始编号的，界限值总是比GDT的总字节数小1。因此，界限值除以8，一定会有余数（余7，丢弃不用），商就是我们所要得到的描述符索引号。最后，将索引号左移3 次，留出TI 位和RPL 位（TI＝0，指向GDT，RPL＝00），这就是要生成的选择子。

第299～306 行，恢复调用之前的现场，返回调用者。返回时用了retf 指令，因此，本过程只能通过远过程调用的方式进入。

继续回到过程load_relocate_program。

安装了用户程序头部段的描述符后，第440 行，将该段的选择子写回到用户程序头部，供用户程序在接管处理器控制权之后使用。实际上，在内核向用户程序转交控制权时，也要用到。

第443～460 行，用于重定位用户程序代码段和数据段，并创建和安装相应的描述符，整个过程都是一样的，也很容易理解。

唯一不同的是栈段，栈所用的空间不需要用户程序提供，而是由内核动态分配。内核分配栈空间时，是以4KB 为单位的，也就是说，每次分配至少是4KB 的倍数。至于到底分配多少，用户程序应该根据自己的实际需求提出建议。

第463 行，从用户程序头部偏移为0x0C 的地方获得一个建议的栈大小。这是一个倍率，至少应当为1，说明用户程序希望分配4KB 栈。如果为2，说明希望分配8KB；为3 则表明希望分配12KB，依此类推。

第464、465 行，计算栈段的界限。如果栈段的粒度是4KB，那么，用0xFFFFF 减去倍率，就是用来创建描述符的段界限。举例来说，如果用户程序建议的倍率是2，那么，这意味着他想创建的栈空间为8KB。因此，段的界限值为

```
0xFFFFF－2＝0xFFFFD
```

那么，当处理器访问该栈段时，实际使用的段界限为

```
0xFFFFD×0x1000＋0xFFF＝0xFFFFDFFF
```

栈是向下扩展的，访问32 位的栈，要使用栈指针寄存器ESP，其最大值是0xFFFFFFFF。因此，ESP 的值只允许在0xFFFFDFFF 和0xFFFFFFFF 之间变化，共8KB。

第466～469 行，用4096（4KB）乘以倍率，得到所需要的栈大小，然后，用这个值去申请内存。这是一个32 位的无符号数乘法，指令格式为

```
mul r/m32
```

这里，用EAX 寄存器的值，乘以另一个32 位的数（可以在通用寄存器或者内存单元里），在EDX:EAX 中得到64 位的乘法结果。

注意，allocate_memory 过程返回所分配内存的低端地址。和一般的数据段不同，栈描述符中的基地址，应当是栈空间的高端地址。所以，第470 行，用allocate_memory 返回的低端地址，加上栈的大小，得到栈空间的高端地址。

第471～473 行，依次调用两个例程，来生成和安装栈段的描述符。注意栈的属性值，它指明了这是一个32 位的栈段，粒度为4KB。

第474 行，将栈段的选择子写回到用户程序头部，供用户程序在接管处理器控制权之后使用。

13.4.5　重定位用户程序内的符号地址

为了使用内核提供的例程，用户程序需要建立一个符号-地址对照表（SALT）。这样，当用户程序加载后，内核应该根据这些符号名来回填它们对应的入口地址，这称为符号地址的重定位。显然，重定位的过程就是字符串匹配和比较的过程。

为了对用户程序内的符号名进行匹配，内核也必须建立一张符号-地址对照表（SALT）。

内核的SALT 表位于代码清单13-2 的内核数据段中，从第338 行开始，一直到第357 行结束。实际上，这个表是可以根据需要扩展的。

如图13-9 所示，用户程序内的SALT 表，每个条目是256 字节，用于容纳符号名，不足256字节的，用零填充。内核中的SALT 表，每个条目则包括两部分，第一部分也是256 字节的符号名；第二部分有6 字节，用于容纳4 字节的偏移地址和2 字节的段选择子，因为符号名是用来描述例程的，这6 字节就是例程的入口地址。

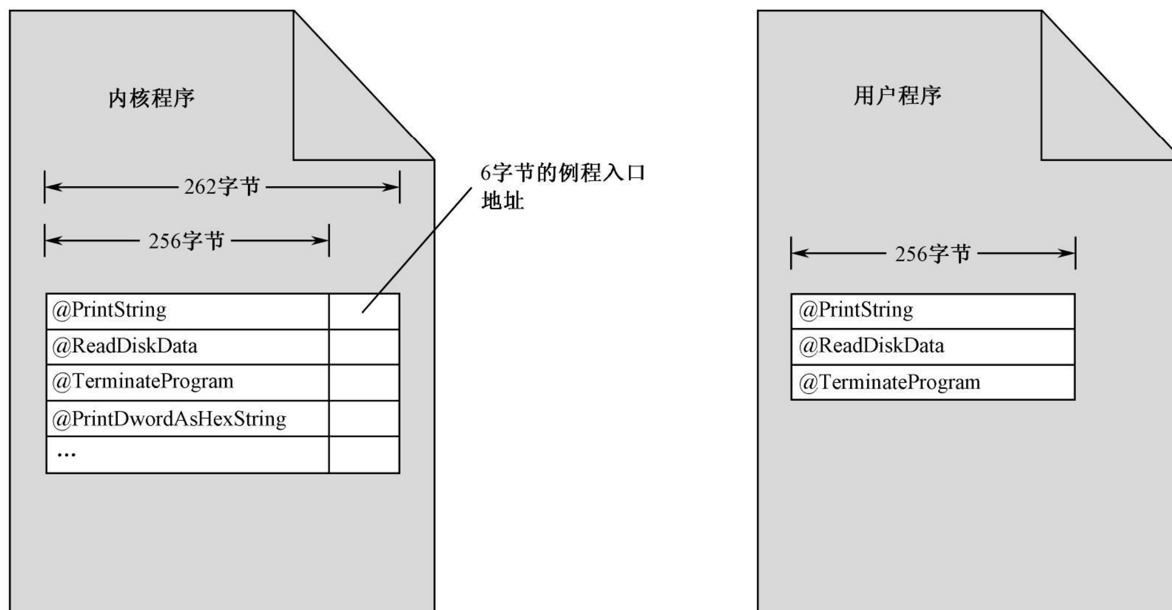


图13-9　内核和用户程序内的符号表结构

举个内核中的例子：

```
salt_1  db  '@PrintString'
        times 256-($-salt_1) db 0
        dd  put_string
        dw  sys_routine_seg_sel
```

这是内核SALT 表的第一个条目。它初始化了一个256 字节的符号名，该名称的前12 个字符是“@PrintString”，因为不足256 字节，后面填充244 个0x00。

在该条目的后面，先是一个双字，初始化为put_string 例程的偏移地址。这就是说，PrintString 其实就是put_string 的别名，调用PrintString，其实是调用put_string 例程。在用户程序内，只能通过远过程调用来进入该例程，所以，该条目的最后是一个字，用公共例程段的选择子来初始化，因为put_string 例程位于公共例程段。

在内核SALT 表中，比较有意思的是最后一个条目：

```
salt_4  db  '@TerminateProgram'
        times 256-($-salt_4) db 0
        dd  return_point
        dw  core_code_seg_sel
```

在这里，从名字可以看出，“TerminateProgram”的意思是终止程序。当用户程序调用该过程时，意味着结束用户程序，将控制返回到内核。

当用户程序终止并返回时，返回点位于标号return_point 所在的位置。该标号位于第582 行，属于内核代码段。在这一行之前，是内核将控制权交给用户程序的指令。

内核的SALT 表是静态的，适用于所有要加载的用户程序，理所当然地要比用户程序的SALT表大，因为它要提供所有可被用户程序调用的过程列表。至于用户程序，根据需要，它只会列出自己用到的那些。

在用户程序加载时，内核的任务是比对这两张SALT 表，并将用户程序SALT 表中的符号名替换成相应的入口地址。为了便于说明，用户程序的SALT 表简称U-SALT，内核的SALT 表简称C-SALT。

基本的算法是使用内外层循环，外循环依次从U-SALT 表中取出条目，每取出一个条目，就进入内循环进行比对；内循环遍历C-SALT 中的每一个条目，同外循环输入的条目进行比对。

比对的过程就是两个字符串的比较过程，可以使用cmps 指令（Compare String Operands）。该指令有3 种基本的形式，分别用于字节、字和双字的比较：

```
        cmpsb               ;字节比较
        cmpsw               ;字比较
        cmpsd               ;双字比较
```

在16 位模式中，源字符串的首地址由DS:SI 指定，目的字符串的首地址由ES:DI 指定；在32位模式下，则分别是DS:ESI 和ES:EDI。在处理器内部，cmps 指令的操作是把两个操作数相减，然后根据结果设置标志寄存器中相应的标志位。

取决于标志寄存器EFLAGS 中的DF 位，如果DF＝0，表明是正向比较，也就是按地址递增的方向比较，这些指令执行后，SI（ESI）和DI（EDI）的内容分别加1、加2 和加4；否则，如果DF ＝1，表明是反向比较，这些指令执行后，SI（ESI）和DI（EDI）的内容分别减1、减2 和减4。

单纯的cmps 指令只比较一次，它属于推一下才动一动的那种类型。所以，需要加指令前缀rep 使比较连续进行。连续比较的次数由CX（ECX）寄存器控制，在16 位模式下，使用CX 寄存器；在32 位模式下，使用ECX 寄存器，举个例子：

```
[bits 32]
rep cmpsd
```

该指令执行时，每次比较4 字节，连续比较直至ECX 寄存器的内容为零。

问题是，用rep 前缀比不出个所以然来，你就是重复比较100000 次，也看不出两个字符串哪里不同。所以，针对cmps 指令，应当使用repe（repz）和repne（repnz）前缀，前者的意思是"若相等（为零）则重复"，后者的意思是"若不等（非零）则重复"。但无论是哪种情况，总的比较次数由CX（ECX）控制，表13-1 显示了这几种控制手段的区别。

表13-1　重复前缀

重 复 前 缀	终止条件一	终止条件二
rep	(E)CX＝0	无
repz/repe	(E)CX＝0	ZF＝0
repnz/repne	(E)CX＝0	ZF＝1

可见，repe/repz 用于搜索第一个不匹配的字节、字或者双字，repne/repnz 用于搜索第一个匹配的字节、字或者双字。无论如何，匹配和不匹配的位置分别由(E)SI 和(E)DI 寄存器指示。

言归正传，我们继续回到代码清单13-2 中来。

如图13-10 所示，为了重定位U-SALT，我们打算用DS:ESI 指向C-SALT，用ES:EDI 指向USALT。第477、478 行，访问4GB 内存段，从用户程序头部偏移为0x04 的地方取出刚刚安装好的头部段选择子，并使段寄存器ES 指向用户程序头部段，因为U-SALT 位于用户程序头部段内。

第479、480 行，使段寄存器DS 指向内核数据段。因为C-SALT 位于内核数据段中。

第482 行，清标志寄存器EFLAGS 中的方向标志，使cmps 指令按正向进行比较。

实施比较的算法我们已经介绍过了。外循环的作用是依次从U-SALT中取出各个条目，因此，第484 行，将取的次数（条目的个数）从用户程序头部取出，传送到ECX 寄存器。

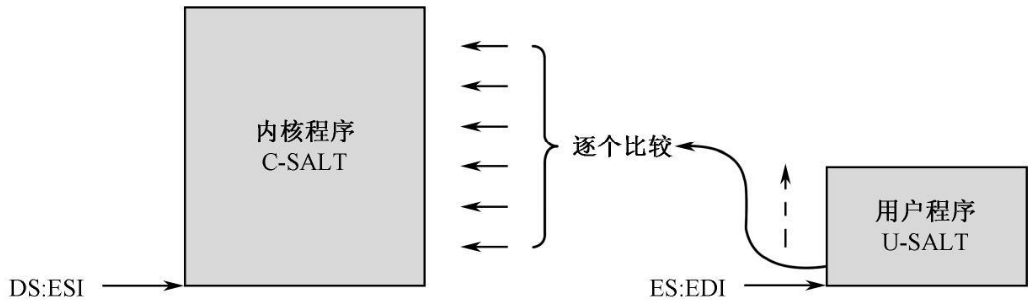


图13-10　U-SALT 和C-SALT 的比对过程示意图

接着，第485 行，用于将U-SALT 在头部段内的偏移量传送到EDI 寄存器。刚才我们已经使段寄存器ES 指向了头部段。

外循环的结构如下所示，这是从代码清单中抽出来的，行号也保持不变。

```
486   .b2:
487         push ecx
488         push edi
489

            ;此处放置内循环代码，用于实际进行比较。

512         pop edi
513         add edi,256
514         pop ecx
515         loop .b2
```

由于内循环也要使用ECX 和EDI 寄存器，并有可能破坏它们的内容，因此，在进入内循环之前，要对它们压栈保护，以便退出内循环后继续使用。外循环的任务是从U-SALT 中依次取出表项，因此，当内循环完成比对后，第512、513 行，从栈中弹出EDI 寄存器的原始内容，并加上256，以指向下一个条目。第514、515 行，从栈中弹出ECX 寄存器的原值。loop 指令将ECX 的内容减一，根据结果判断是否继续循环。

对于外循环所指向的每一个条目，内循环要用它和C-SALT 中的所有条目进行比对，内循环的代码如下：

```
490         mov ecx,salt_items
491         mov esi,salt
492   .b3:
493         push edi
494         push esi
495         push ecx

              ;这里放置实际进行比对的代码

506         pop ecx
507         pop esi
508         add esi,salt_item_len
509         pop edi
510         loop .b3
```

每次从外循环进入内循环时，都要重新设置比对次数，并重新使ESI 寄存器指向C-SALT 的开始处，这是第490、491 行的工作。标号salt_item_len 是在第359 行声明的，并用一个表达式初始化。每个条目的长度都是相同的，用当前汇编地址减去标号salt_4 的汇编地址，即$-salt_4，就是每个条目的长度（字节数）。事实上，这个数值是在编译阶段由编译器计算的，在数值上等于262。

标号salt_items 是在第360 行声明的，并初始化为一个表达式。该表达式的意思是，用整个CSALT 的长度，除以每个条目的长度，就是条目的个数。

对于内循环的每一次执行，都要把ESI、EDI 和ECX 压栈保护，以免在比对的过程中用到并破坏这些寄存器。每次比对结束后，第506～509 行，依次弹出这些寄存器的值，并把ESI 的内容加上C-SALT 每个条目的长度（262 字节），以指向下一个C-SALT 条目。第510 行，loop 指令执行时，将ECX 的内容减一并判断是否继续循环。

第497～503 行，是整个比对过程的核心部分。每当处理器执行到这里时，DS:ESI 和ES:EDI都各自指向C-SALT 和U-SALT 中的某个条目：

```
497         mov ecx,64
498         repe cmpsd
499         jnz .b4
500         mov eax,[esi]
501         mov [es:edi-256],eax
502         mov ax,[esi+4]
503         mov [es:edi-252],ax
504   .b4:
```

因为每个条目的符号名部分是256 字节，每次用cmpsd 指令比较4 字节，故每个条目至多需要比对64 次。第497 行把立即数64 传送到ECX 寄存器以控制整个比对过程。

第498 行，开始比对，直到发现一个不相符的地方。

如果两个字符串相同，则需要连续比对64 次，而且，在比对结束时，ZF＝1，表示最后4字节也相同；如果两个字符串不同，比对过程会提前结束，且ZF＝0。在最坏的情况下，这两个字符串可能只有最后4 字节是不同的。在这种情况下，也需要比对64 次，但ZF＝0。

无论哪种情况，如果在退出repe cmpsd 指令时ZF＝0，即表明两个字符串是不同的。所以，第499 行，如果ZF＝0，则表明两个字符串不同，直接转移到内循环的末尾，以开始下一次内循环。

如果两个字符串是相同的，那么，比较指令执行后，ESI 寄存器正好指向C-SALT 每个条目后的入口数据。要知道，C-SALT 中的每个条目是262 字节，最后的6 字节分别是偏移地址和段选择子。

因此，现在的任务是将这结尾的6 字节传送到U-SALT 当前条目的开始部分，这是第500～503 行的工作。最后的结果是，U-SALT 中的当前条目，其开始的6 字节被改写为一个入口地址。

13.5 执行用户程序

在load_relocate_program 过程的最后，第517 行，把用户程序头部段的选择子传送到AX 寄存器。第519～528 行，从栈中弹出并恢复各个寄存器的原始内容，并返回到调用者。AX 寄存器中的选择子是作为参数返回到主程序的。主程序将用它来找到用户程序的入口，并从那里进入。

从load_relocate_program 过程返回后，第572、573 行用于在屏幕上显示信息，表示加载和重定位工作已经完成。

第575 行，保存内核的栈指针。这是通过将ESP 寄存器的当前值写入内核数据段中来完成的。写入的位置是由标号esp_pointer 指示的，位于第378 行，初始化为一个双字。在进入用户程序后，用户程序应当切换到它自己的栈。从用户程序返回时，还要从这个内存位置还原内核栈指针。

第577 行，使段寄存器DS 指向用户程序头部。这是通过将用户程序头部段选择子传送到DS 来办到的。在用户程序头部段内偏移0x10 处，是用户程序的入口点，分别是32 位的偏移量和16 位的代码段选择子。第579 行，执行一个间接远转移，进入用户程序内接着执行。

现在转到代码清单13-3。

用户程序的入口点是在第56 行。进入用户程序开始执行时，段寄存器DS 是指向头部段的。第57、58 行，使段寄存器FS 指向头部段，因为后面要调用内核过程，而这些过程都要求使用DS，所以要把DS 解放出来。

第60～62 行，切换到用户程序自己的栈，并初始化栈指针寄存器ESP 的内容为0。

第64、65 行，设置段寄存器DS 到用户程序自己的数据段。

第67、68 行，调用内核过程显示字符串，以表明用户程序正在运行中。该内核过程要求用DS:EBX 指向零终止的字符串。

第70～72 行，调用内核过程，从硬盘读一个扇区。从内核代码清单可以知道，ReadDiskData过程的内部名称是read_hard_disk_0。所以，

ReadDiskData 需要传入两个参数，第一个是EAX 寄存器，传入要读的逻辑扇区号；第二个是DS:EBX，传入缓冲区的首地址，毕竟读出来的数据要有个地方保存。缓冲区位于用户程序的数据段中，是在第43 行用标号buffer 声明的，并初始化了1024 字节的空间。要读的逻辑扇区号是100，在此之前，我们应当在这个扇区里写一些东西。这件事我们马上就要讲到。

第74～78 行，先调用内核过程显示一个题头，接着，再次调用内核过程显示刚刚从硬盘读出的内容。

在做完了上述事情之后，用户程序的任务也就完成了。第80 行，调用内核过程，以返回到内核。

再次回到代码清单13-2。

在内核中，用户程序的返回点位于第582 行。

在重新接管了处理器的控制权后，第583、584 行，使段寄存器DS 重新指向内核数据段。

第586～588 行，切换栈，使栈段寄存器SS 重新指向内核栈段，并从内核数据段中取得和恢复原先的栈指针位置。

第590、591 行，显示一条消息，表示现在已经回到了内核。

对于一个操作系统来说，下面的任务是回收前一个用户程序所占用的内存，并启动下一个用户程序。但是，我们现在无事可做，所以，第596 行，使处理器进入停机状态。别忘了，在进入保护模式之前，我们已经用cli 指令关闭了中断，所以，除非有NMI 产生，处理器将一直处于停机状态。

13.6 代码的编译、运行和调试

首先编译本章所有的源程序文件，它们是c13_mbr.asm、c13_core.asm 和c13.asm，这将分别生成c13_mbr.bin、c13_core.bin 以及c13.bin。

使用配书工具FixVhdWr 分别将这些二进制文件写入虚拟硬盘。c13_mbr.bin 的起始逻辑扇区号是0，因为它是主引导代码；c13_core.bin 的起始逻辑扇区号是1；c13.bin 的起始逻辑扇区号是50。除了c13_mbr.bin 外，其他文件的写入位置可以改变，但前提是要修改使用它们的源代码。

用户程序的功能是读取逻辑扇区100，并显示其内容。为此，需要找一个文本文件，并将它写入该扇区。在配书源代码中，提供了一个文本文件diskdata.txt，其大小是512 字节。如图13-11 所示，它包含了512 字节的英文文本。

不强迫你一定要使用这个文件。你完全可以选用其他文件，文件的内容也无所谓，但最好是可读的ASCII 字符。

使用配书工具FixVhdWr 将你采用的文本文件写入虚拟硬盘，逻辑扇区号是100。如果你采用的是其他文件，它或许很长，会连续写入多个扇区。这无所谓，用户程序只读取第一个。

最后，启动虚拟机时，如果一切正常，所显示的画面将如图13-12 所示。

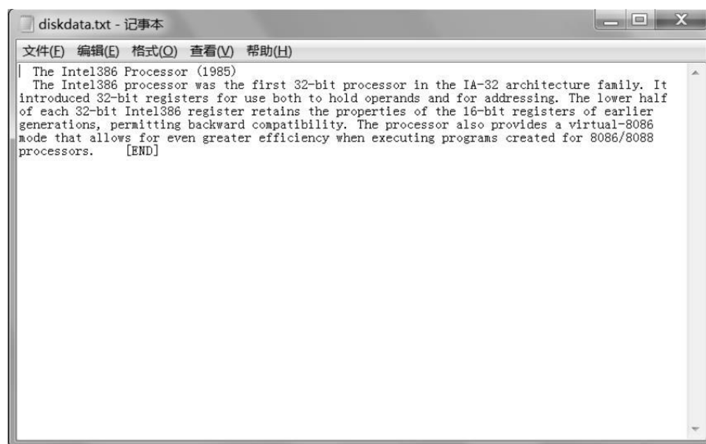


图13-11 diskdata.txt 文件的内容



图13-12 本章程序的运行结果

具有讽刺意味的是，我在这里大书特书、侃侃而谈INTEL 的处理器，但是，从截图上可以看出，我用的处理器却是AMD 生产的。至于你的计算机用了什么处理器，你自己看看吧，屏幕上的显示会说明一切的。

随着程序代码量的增大，程序的编写和调试也会变得越来越困难。特别是当问题发生的时候，追查出错的位置和错误的原因都需要花费大量的时间、消耗大量的精力。

有时候，最简单的方法却很有效。比如，可以写一个特殊的过程，用来显示某个寄存器的内容。如果你的程序运行时出了问题，可以在有重大嫌疑的指令前后安排一些调用该过程的代码，看看是哪里不正常。这些用于调试程序的位置，叫做检查点。

为了方便调试程序，代码清单13-2 提供了一个过程put_hex_dword，用于以十六进制的形式显示EDX 寄存器的内容。

该过程位于第202 行，它的工作原理很简单，EDX 寄存器是32 位的，从右到左，将它以4 位为一组，分成8 组。每一组的值都在0～15（0x0～0xf）之间，我们把它转换成相应的字符'0' ～'F'即可。

为了将数值转换成可显示的ASCII 码，可以使用处理器的查表指令xlat（Table Look-up Translation），该指令要求事先在DS:(E)BX 处定义一个用于转换编码的表格，在16 位模式下，使用BX 寄存器；在32 位模式下，使用EBX 寄存器。指令执行时，处理器访问该表格，用AL 寄存器的内容作为偏移量，从表格中取出一字节，传回AL 寄存器。

代码清单13-2 定义的表格在第374 行。在那里，声明了标号bin_hex，并初始化了16 个字符，这是一个二进制到十六进制的对照（检索）表。偏移（索引）为0 的位置是字符“0”；偏移（索引）为0x0f 的位置是字符“F”。

第209、210 行，使段寄存器DS 指向内核数据段，因为对照表bin_hex 位于内核数据段中。

第212 行，使EBX 寄存器指向检索（对照表）的起始处。

转换过程使用了循环，每次将EDX 寄存器的内容循环左移4 位，共需要循环8 次。每次移位后的内容被传送到EAX 寄存器，并用and 指令保留低4 位，高位清零。第218 行，xlat 指令用AL 寄存器中的值作为索引访问对照表，取出相应的字符，并回传到AL 寄存器。

每次从检索（对照）表中得到一个字符，就要调用put_char 过程显示它。但put_char 过程需要使用CL 寄存器作为参数。因此，第220 行，在显示之前先要将ECX 寄存器压栈保护。

xlat指令不影响任何标志位。

本章习题

在本章中，用户程序只给出建议的栈大小，但并不提供栈空间。现在，修改内核程序和用户程序，改由用户程序自行提供栈空间。要求：栈段必须定义在用户程序头部之后。

第14章　任务和特权级保护

在保护模式下，通过将内存分成大小不等的段，并用描述符对每个段的用途、类型和长度进行指定，就可以在程序运行时由处理器硬件施加访问保护。比如，当程序试图让处理器去写一个可执行的代码段时，处理器就会阻止这种企图；再比如，当程序试图让处理器访问超过段界限的内存区域时，处理器也会引发异常中断。

段保护是处理器提供的基本保护功能，但对于现实的需求来说，仍是不够的。

首先，当一个程序老老实实地访问只属于它自己的段时，基本的段保护机制是很有效的。但是，一个失控的程序，或者一个恶意的程序，依然可以通过追踪和修改描述符表来达到它们访问任何内存位置的目的。比如说，如果用户程序知道GDT 的位置，它可以通过向段寄存器加载操作系统的数据段描述符，或者在GDT 中增加一个指向操作系统数据区的描述符，来修改只属于操作系统的私有数据。对于处理器那种和3 岁小孩相仿的智力，所有这一切都是合法的。

其次，32 位处理器是为多任务系统而设计的。所谓多任务系统，是指能够同时执行两个以上程序的系统，即使前一个程序没有执行完，其他程序也可以开始执行。在单处理器（核）的系统中，多个程序并不可能真的同时执行，但是，处理器可以在多个任务之间周期性地切换和轮转。这样，它们都处于走走停停的状态，快速的处理器加上高效的任务切换，在外界看来，多个任务都在同时运行中。

多任务系统，对任务之间的隔离和保护，以及任务和操作系统之间的隔离和保护都提出了要求，这可以看做对段保护机制的进一步强化。同时，在多任务系统中，操作系统居于核心软件的位置，为各个任务服务，负责任务的加载、创建和执行环境的管理，并执行任务之间的调度，对操作系统的保护显得尤为重要。事实上，对于这种要求，基本的段保护机制已经无能为力了。

综上所述，本章的学习目标是：

1．通过演示如何创建一个任务，并使之投入运行来学习任务的概念及其组成要素，包括任务的全局空间和局部空间、TSS、LDT、特权级等。

2．必须了解特别级不是指任务的特权级，而是指组成任务的各个部分的特权级。比如，任务的全局部分一般是0、1 和2 特权级别的，任务的私有部分一般是3 特权级别的。

3．必须清楚CPL、DPL 和RPL 的含义，以及不同特权级别之间的控制转移规则。

4．熟悉调用门的用法。

5．掌握一些在Bochs 下调试程序的新手段。

6．学习一些新的x86 处理器指令，包括lldt、ltr、pushf/pushfd、popf/popfd、ret n/retf n、arpl等，同时，了解象jmp 和call 这样的传统指令是如何被赋予一些新功能的。

14.1　任务的隔离和特权级保护

14.1.1　任务、任务的LDT 和TSS

程序（Program）是记录在载体上的指令和数据，总是为了完成某个特定的工作，其正在执行中的一个副本，叫做任务（Task）。这句话的意思是说，如果一个程序有多个副本正在内存中运行，那么，它对应着多个任务，每一个副本都是一个任务。在上一章里，用户程序就是任务，而内核程序就是操作系统的缩影。

一直以来，我们把所有的段描述符都放在GDT 中，而不管它属于内核还是用户程序。如图14-1 所示，为了有效地在任务之间实施隔离，处理器建议每个任务都应当具有自己的描述符表，称为局部描述符表LDT（Local Descriptor Table），并且把专属于自己的那些段放到LDT中。

和GDT 一样，LDT 也是用来存放描述符的。不同之处在于，LDT 只属于某个任务。或者说，每个任务都有自己的LDT，每个任务私有的段，都应当在LDT 中进行描述。另外，LDT 的第1 个描述符，也就是0号槽位，也是有效的、可以使用的。

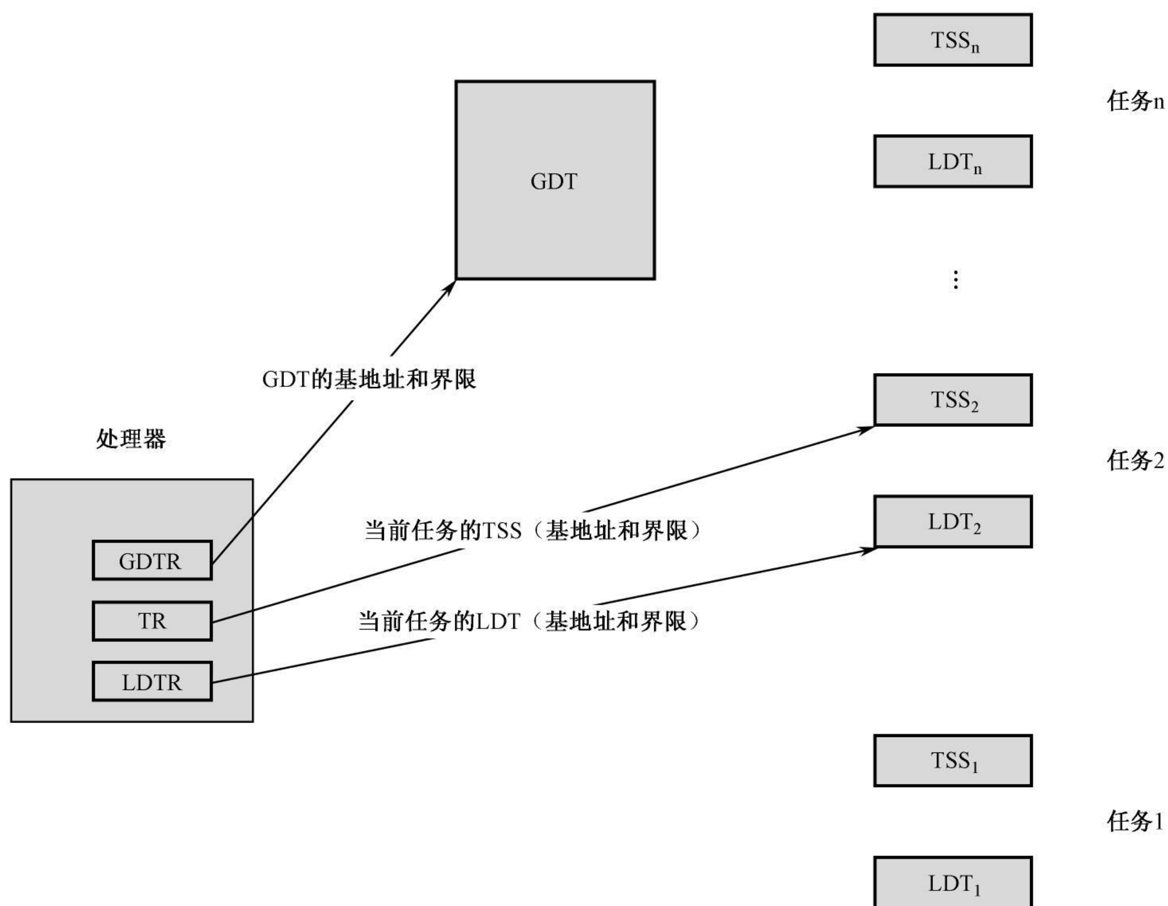


图14-1　多任务系统的组成示意图

为了追踪全局描述符表（GDT），访问它内部的描述符，处理器使用了GDTR 寄存器。这是可以理解的，正如其名称所暗示的那样，全局描述符表（GDT）是全局性的，为所有任务服务，是它们所共有的，我们只需要一个全局描述符表（GDT）就够了。

和GDT 不同，局部描述符表（LDT）的数量则不止一个，具体有多少，视任务的多少而定。为了追踪和访问这些LDT，处理器使用了局部描述符表寄存器（LDT Register：LDTR）。

在一个多任务的系统中，会有很多任务在轮流执行，正在执行中的那个任务，称为当前任务（Current Task）。因为LDTR 寄存器只有一个，所以，它只用于指向当前任务的LDT。每当发生任务切换时，LDTR 的内容被更新，以指向新任务的LDT。和GDTR 一样，LDTR 包含了32 位线性基地址字段和16 位段界限字段，以指示当前LDT 的位置和大小。

我们知道，在访问内存之前需要先指定一个段，方法是向段寄存器的选择器传送一个段选择子，这称为“引用一个段”，像这样：

```
mov cx,0x0008
mov ds,cx
```

回到第11 章，看一下图11-10，段选择子的位2 是表指示器（Table Indicator：TI），若TI＝0，表示从GDT 中加载描述符；TI＝1，表示从当前任务的LDT 中加载描述符。

很显然，0x0008 的二进制形式为0000 0000 0000 1000，其TI 位是“0”，所以，处理器将访问GDT，从1 号槽位取得描述符，并传送到段寄存器DS 的描述符高速缓存器。

再看这个例子：

```
mov cx,0x005c
mov ds,cx
```

0x005C 的二进制形式为0000 0000 0101 1100，这很容易看出TI 位是“1”，索引号为11（十进制）。处理器执行以上指令时，必然会访问当前任务的LDT（该LDT 在内存中的位置由LDTR指定），从它的11 号槽位取出描述符，并传送到段寄存器DS 的描述符高速缓存器中去。

很显然，因为段选择子是16 位的，而且只有高13 位被用做索引号来访问GDT 或者LDT，所以，每个LDT 所能容纳的描述符个数为213，即8192 个。或者换句话说，每个LDT 只能定义8192 个段。又因为每个描述符的长度是8 字节，LDT 的长度最大为64KB。

在一个多任务的环境中，当任务切换发生时，必须保护旧任务的运行状态，或者说是保护现场，保护的内容包括通用寄存器、段寄存器、栈指针寄存器ESP、指令指针寄存器EIP、状态寄存器EFLAGS，等等。否则的话，等下次该任务又恢复执行时，一切都会变得茫然而毫无头绪。

为了保存任务的状态，并在下次重新执行时恢复它们，每个任务都应当用一个额外的内存区域保存相关信息，这叫做任务状态段（Task State Segment：TSS）。如图14-2 所示，任务状态段TSS 具有固定的格式，最小尺寸是104 字节，**图中所标注的偏移量是十进制的** 。处理

器固件能够识别TSS 中的每个元素，并在任务切换的时候读取其中的信息，具体的细节将在后面讲述。

和LDT 一样，处理器用TR 寄存器来指向当前任务的TSS。和GDTR、LDTR 一样，TR 寄存器在处理器中也只有一个。当任务切换发生的时候，TR 寄存器的内容也会跟着指向新任务的TSS。这个过程是这样的：首先，处理器将当前任务的现场信息保存到由TR 寄存器指向的TSS；然后，再使TR 寄存器指向新任务的TSS，并从新任务的TSS 中恢复现场。

比较奇怪的是，为什么这个寄存器叫TR，而不是TSSR。原因很简单，TSS 是一个任务存在的标志，用于区别一个任务和其他任务。所以，这个寄存器叫做任务寄存器（Task Register：TR）。

31	15	0	
I/O映射基地址	（保留）	T	100
（保留）	LDT段选择子		96
（保留）	GS		92
（保留）	FS		88
（保留）	DS		84
（保留）	SS		80
（保留）	CS		76
（保留）	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3(PDBR)			28
（保留）	SS2		24
ESP2			20
（保留）	SS1		16
ESP1			12
（保留）	SS0		8
ESP0			4
（保留）	前一个任务的指针（TSS）		0

图14-2　32 位的任务状态段

14.1.2　全局空间和局部空间

现代的计算机，如果没有操作系统支持，它也可以在编程爱好者的操作下运行得很好，但恐怕不太可能像比尔·盖茨所认为的那样，每个桌子上一台。

在多任务系统中，操作系统肩负着任务的创建，以及在任务之间调度和切换的工作。不过，更为繁重和基础的工作是对处理器、设备及存储器的管理。

从程序编写者的角度看，操作系统是他们可以信赖的朋友。首先，他们不必关心自己的程序是如何加载到内存并开始运行的，操作系统自然会处理好这些事情；其次，对设备的访问涉及大量的硬件细节，而且极为烦琐，操作系统能够肩负起设备管理的职责，并提供大量的例程和数据供应用程序调用。使用操作系统提供的这些服务，可以极大地简化程序的编写，并能够在访问设备时消除潜在的竞争和冲突。

比如说，当中断发生时，不可能由某个任务来进行处理，而只能由操作系统来提供中断处理过程，并采取适当的操作，以进行一些和所有任务都有关系的全局性管理工作，如空闲内存的查找和分配、回收已终止任务的内存空间、设备访问的排队和调度，等等。

这就是说，如图14-3 所示，每个任务实际上包括两个部分：全局部分和私有部分。全局部分是所有任务共有的，含有操作系统的软件和库程序，以及可以调用的系统服务和数据；私有部分则是每个任务各自的数据和代码，与任务所要解决的具体问题有关，彼此并不相同。

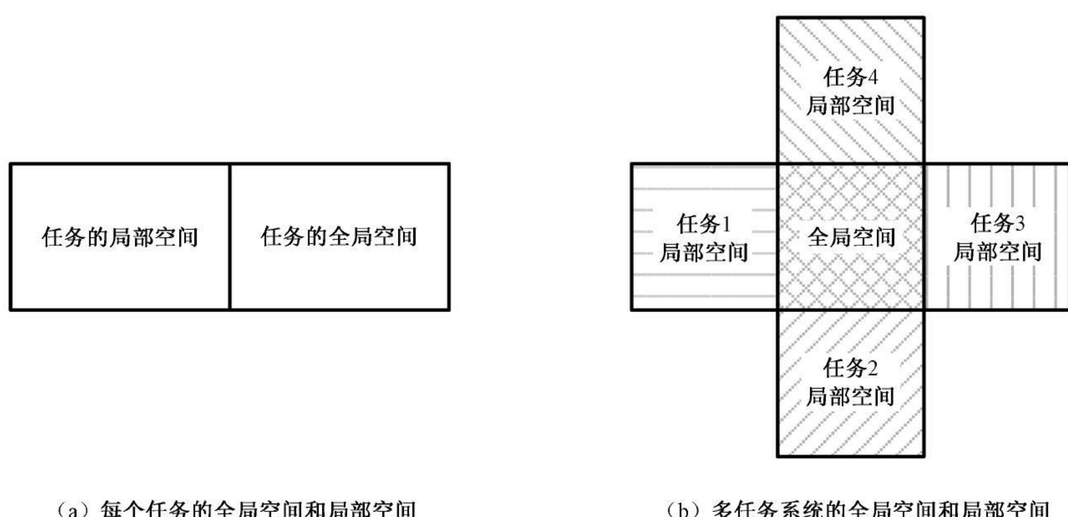


图14-3　任务的全局空间和局部空间

任务实际上是在内存中运行的，所以，所谓的全局部分和私有部分，其实是地址空间的划分，即全局地址空间和局部地址空间，简称全局空间和局部空间。

地址空间的访问是依靠分段机制来进行的。具体地说，需要先在描述符表中定义各个段的描述符，然后再通过描述符来访问它们。因此，全局地址空间是用全局描述符表（GDT）来指定的，而局部地址空间则是由每个任务私有的局部描述符表（LDT）来定义的。

从程序员的角度来看，任务的全局空间包含了操作系统的段，是由别人编写的，但是他可以调用这些段的代码，或者获取这些段中的数据；任务局部空间的内容是由程序员自己创建的。通常，任务会在自己的局部空间运行，当它需要操作系统提供的服务时，转入全局空间执行。

我们知道，段寄存器（CS、SS、DS、ES、FS 和GS）由16 位的选择器和不可见的描述符高速缓存器组成。选择器的位2 是表指示器TI，若TI＝0，指向GDT，表示当前正在访问的段描述符位于GDT 中；否则指向LDT，表示当前正在访问的段描述符位于LDT 中。选择器的高13 位指定描述符的索引号，也就是描述符在描述符表中的编号，从0 开始。

每个段描述符都对应着一个内存段。很显然，在一个任务的全局地址空间上，可以划分出2^{13} 个段，也就是8192 个段。因为GDT 的0 号描述符不能使用，故实际上是8191 个段，但这无关紧要。又因为段内偏移是32 位的，段的长度最大的4GB，因此，一个任务的全局地址空间，其总大小为$2^{13} \times 2^{32} = 2^{45}$ 字节，即32TB。

同样的道理，局部描述符表LDT 可以定义2^{13} 个，也就是8192 个描述符，每个段的最大长度也是4GB，故，一个任务的局部地址空间为$2^{13} \times 2^{32} = 2^{45}$ 字节，同样是32TB。

这样一来，每个任务的总地址空间为$2^{45} + 2^{45} = 2^{45} \times 2 = 2^{45} \times 2^{1} = 2^{46}$ 字节，即64TB。在一个只有32 根地址线的处理器上，无论如何也不可能提供这样巨大的存储空间，但是，不要紧张，这只是虚假的，或者说虚拟的地址空间。操作系统允许程序的编写者使用该地址空间来写程序，即，使用虚拟地址或者逻辑地址来访问内存，就像他真的拥有这么巨大的地址空间一样。

上面一段话可以这样理解：编译器不考虑处理器可寻址空间的大小，也不考虑物理内存的大小，它只是负责编译程序。当程序编译时，编译器允许生成非常巨大的程序。但是，当程序超出了物理内存的大小时，或者操作系统无法分配这么大的物理内存空间时，怎么办呢？

同一块物理内存，可以让多个任务，或者每个任务的不同段来使用。当执行或者访问一个新的段时，如果它不在物理内存中，而且也没有空闲的物理内存空间来加载它，那么，操作系统将挑出一个暂时用不到的段，把它换出到磁盘中，并把那个腾出来的空间分配给马上要访问

的段，并修改段的描述符，使之指向这段内存空间。下一次，当被换出的那个段马上又要用到时，再按相同的办法换回到物理内存。所有这一切，任务（如果它有思维的话）和程序的编写者是不必关心的，这就是虚拟内存管理的一般方法。

14.1.3 特权级保护概述

引入LDT 和TSS，只是从任务层面上进一步强化了分段机制，从安全保障的角度来看，只相当于构建了可靠的硬件设施。

当然，仅有设施是不够的，还需要规章制度，还要有人来执行，处理器也一样。为此，在分段机制的基础上，处理器引入了特权级，并由固件负责实施特权级保护。

特权级（Privilege Level），也叫特权级别，是存在于描述符及其选择子中的一个数值，当这些描述符或者选择子所指向的对象要进行某种操作，或者被别的对象访问时，该数值用于控制它们所能进行的操作，或者限制它们的可访问性。

Intel 处理器可以识别4 个特权级别，分别是0 到3，较大的数值意味着较低的特权级别，反之亦然。如图14-4 所示，这是Intel 处理器所提供的4 级环状保护结构。

通常，因为操作系统是为所有程序服务的，可靠性最高，而且必须对软硬件有完全的控制权，所以它的主体部分必须拥有特权级0，并处于整个环形结构的中心。也正是因为这样，操作系统的主体部分通常又被称做内核（Kernel、Core）。

特权级1 和2 通常赋予那些可靠性不如内核的系统服务程序，比较典型的就是设备驱动程序。当然，在很多比较流行的操作系统中，驱动程序与内核的特权级别相同，都是0。

应用程序的可靠性被视为是最低的，而且通常不需要直接访问硬件和一些敏感的系统资源，调用设备驱动程序或者操作系统例程就能完成绝大多数工作，故赋予它们最低的特权级别3。

实施特权级保护的第一步，是为所有可管理的对象赋予一个特权级，以决定谁能访问它们。回到第11 章，看图11-4。图中，每个描述符都有一个两比特的DPL 字段，可以取值为00、01、10 和11，分别对应特

权级0、1、2 和3。DPL 是每个描述符都有的字段，故又称描述符特权级（Descriptor Privilege Level）。描述符总是指向它所描述的目标对象，代表着该对象，因此，该字段实际上是目标对象的特权级。

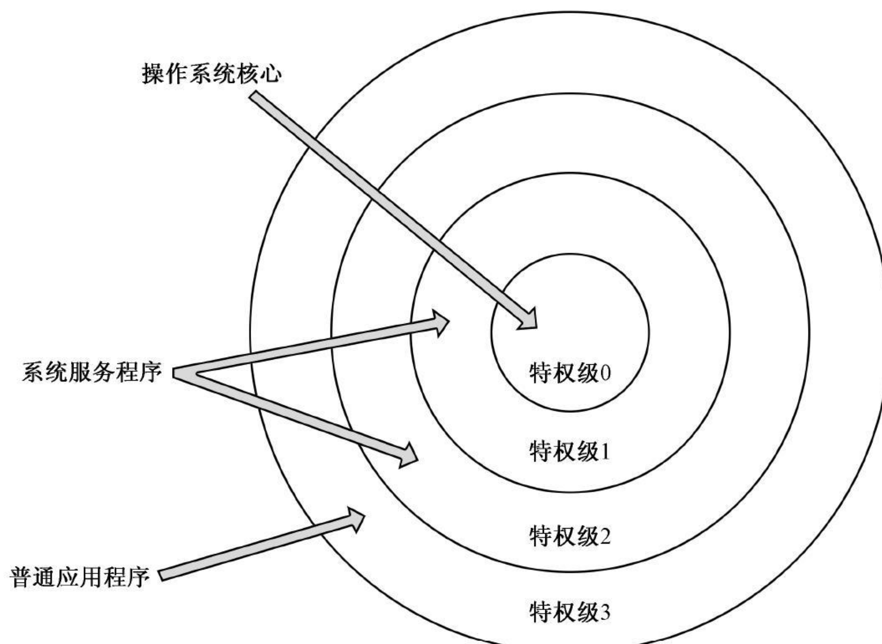


图14-4 处理器的4 级环状保护结构

比如，对于数据段来说，DPL 决定了访问它们所应当具备的最低特权级别。如果有一个数据段，其描述符的DPL 字段为2，那么，只有特权级为0、1 和2 的程序才能访问它。当一个特权级为3 的程序也试图去读写该段时，将会被处理器阻止，并引发异常中断。对任何段的访问都要先把它的描述符加载到段寄存器，所以这种保护手段很容易实现。

我们知道，32 位处理器的段寄存器，实际上由16 位的段选择器和描述符高速缓存器组成，而且后者是不能直接访问的。正因为我们接触不到描述符高速缓存器，所以，为了方便，以后我们提到段寄存器的时候，指的就是段选择器。

在实模式下，段寄存器存放的是段地址；而在保护模式下，段寄存器存放的是段选择子，段地址则位于描述符高速缓存器中。当处理器正在一个代码段中取指令和执行指令时，那个代码段的特权级叫做当前特权级（Current Privilege Level，CPL）。正在执行的这个代码段，其选择子位于段寄存器CS 中，其最低两位就是当前特权级的数值。

一般来说，操作系统是最先从BIOS 那里接收处理器控制权的，进入保护模式的工作也是由它做的，而且，最重要的是，它还肩负着整个计算机系统的管理工作，所以，它必须工作在0 特权级别上，当操作系统的代码正在执行时，当前特权级CPL 就是0。

相反，普通的应用程序则工作在特权级别3 上。没有人愿意将自己的程序放在特权级3 上，但是，只要你在某个操作系统上面写程序，这就由不得你。应用程序编写时，不需要考虑GDT、LDT、分段、描述符这些东西，它们是在程序加载时，由操作系统负责创建的，应用程序的编写者只负责具体的功能就可以了。应用程序的加载和开始执行，也是由操作系统所主导的，而操作系统一定会将它放在特权级3 上。当应用程序开始执行时，当前特权级CPL 自然就会是3。

这实际上就是把一个任务分成特权级截然不同的两个部分，全局部分是特权级0 的，而局部空间则是特权级3 的。这种划分是有好处的，全局空间是为所有任务服务的，其重要性不言而喻。为了保证它的安全性，并能够访问所有软硬件资源，应该使它拥有最高的特权级别。当任务在自己的局部空间内执行时，当前特权级CPL 是3；当它通过调用系统服务，进入操作系统内核，在全局空间执行时，当前特权级CPL 就变成了0。总之，很重要的一点是，不能僵化地看待任务和任务的特权级别。

不同特权级别的程序，所担负的职责以及在系统中扮演的角色是不一样的。计算机系统的脆弱性在于一条指令就能改变它的整体运行状态，比如停机指令hlt 和对控制寄存器CR0 的写操作，像这样的指令只能由最高特权级别的程序来做。因此，那些只有在当前特权级CPL 为0 时才能执行的指令，称为特权指令（Privileged Instructions）。典型的特权指令包括加载全局描述符表的指令lgdt（它在实模式下也可执行）、加载局部描述符表的指令lldt、加载任务寄存器的指令ltr、读写控制寄存器的mov 指令、停机指令hlt 等十几条。

除了那些特权级敏感的指令外，处理器还允许对各个特权级别所能执行的I/O 操作进行控制。通常，这指的是端口访问的许可权，因为对设备的访问都是通过端口进行的。如图14-5 所示，在处理器的标志寄存器EFLAGS 中，位13、位12 是IOPL 位，也就是输入/输出特权级（I/O Privilege Level），它代表着当前任务的I/O 特权级别。

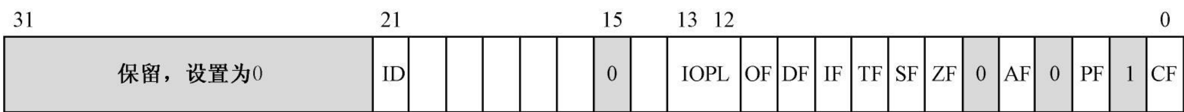


图14-5　EFLAGS 寄存器中的IOPL 位

任务是由操作系统加载和创建的，与任务相关的信息都在它自己的任务状态段（TSS）中，其中就包括一个EFLAGS 寄存器的副本，用于指示与当前任务相关的机器状态，比如它自己的I/O特权级IOPL。在多任务系统中，随着任务的切换，前一个任务的所有状态被保存到它自己的TSS中，新任务的各种状态从其TSS 中恢复，包括EFLAGS 寄存器的值。

处理器不限制0 特权级程序的I/O 访问，它总是允许的。但是，可以限制低特权级程序的I/O访问权限。这是很重要的，操作系统的功能之一是设备管理，它可能不希望应用程序拥有私自访问外设的能力。

代码段的特权级检查是很严格的。一般来说，控制转移只允许发生在两个特权级相同的代码段之间。如果当前特权级为2，那么，它可以转移到另一个DPL 为2 的代码段接着执行，但不允许转移到DPL 为0、1 和3 的代码段执行。不过，为了让特权级低的应用程序可以调用特权级高的操作系统例程，处理器也提供了相应的解决办法。

第一种方法是将高特权级的代码段定义为依从的。回到第11 章，在那一章里，表11-1 给出了段描述符的TYPE 字段。代码段描述符的TYPE 字段有C 位，如果C＝0，这样的代码段只能供同特权级的程序使用；否则，如果C＝1，则这样的代码段称为依从的代码段，可以从特权级比它低的程序调用并进入。

但是，即使是将控制转移到依从的代码段，也是有条件的，要求当前特权级CPL 必须低于，或者和目标代码段描述符的DPL 相同。即，在数值上，

```
CPL≥目标代码段描述符的 DPL
```

举例来说，如果一个依从的代码段，其描述符的DPL 为1，则只有特权级别为1、2、3 的程序可以调用，而特权级为0 的程序则不能。在任何时候，都不允许将控制从较高的特权级转移到较低的特权级。

依从的代码段不是在它的DPL 特权级上运行，而是在调用程序的特权级上运行。就是说，当控制转移到依从的代码段上执行时，不改变当前特权级CPL，段寄存器CS 的CPL 字段不发生变化，被调用过程的特权级依从于调用者的特权级，这就是为什么它被称为“依从的”代码段。

除了依从的代码段，另一种在特权级之间转移控制的方法是使用门。门（Gate）是另一种形式的描述符，称为门描述符，简称门。和段描述符不同，段描述符用于描述内存段，门描述符则用于描述可执行的代码，比如一段程序、一个过程（例程）或者一个任务。

事际上，根据不同的用途，门的类型有好几种。不同特权级之间的过程调用可以使用调用门；中断门/陷阱门是作为中断处理过程使用的；任务门对应着单个的任务，用来执行任务切换。在本章里，我们重点介绍的是调用门（Call Gate）。

所有描述符都是64 位的，调用门描述符也不例外。在调用门描述符中，定义了目标过程（例程）所在代码段的选择子，以及段内偏移。要想通过调用门进行控制转移，可以使用jmp far 或者call far 指令，并把调用门描述符的选择子作为操作数。

使用jmp far 指令，可以将控制通过门转移到比当前特权级高的代码段，但不改变当前特权级别。但是，如果使用call far 指令，则当前特权级会提升到目标代码段的特权级别。也就是说，处理器是在目标代码段的特权级上执行的。但是，除了从高特权级别的例程（通常是操作系统例程）返回外，不允许从特权级高的代码段将控制转移到特权级低的代码段，因为操作系统不会引用可靠性比自己低的代码。

说了这么多，好像这是我们头一回接触特权级似的。

事实上，它是老朋友了，从第11 章我们写第一个保护模式程序开始，我们就在创建DPL 为0的描述符，只不过从来没有向大家介绍。远的就不说了，就说上一章，也就是第13 章，这一章比较典型，既有内核程序，也有用户程序（应用程序）。

参见代码清单13-1，也就是源程序c13_mbr.asm，第24～37 行，创建了初始的几个段描述符：

```
;创建 1#描述符，这是一个数据段，对应 0~4GB 的线性地址空间
mov dword [ebx+0x08],0x0000ffff         ;基地址为 0，段界限为 0xFFFFF
```

```
mov dword [ebx+0x0c],0x00cf9200          ;粒度为 4KB，数据段，DPL=00

;创建保护模式下初始代码段描述符
mov dword [ebx+0x10],0x7c0001ff          ;基地址为 0x7C00，界限 0x1FF
mov dword [ebx+0x14],0x00409800          ;粒度为字节，代码段，DPL=00

;建立保护模式下的栈段描述符
mov dword [ebx+0x18],0x7c00fffe          ;基地址为 0x7C00，界限 0xFFFFE
mov dword [ebx+0x1c],0x00cf9600          ;粒度为 4KB，栈段，DPL=00

;建立保护模式下的显示缓冲区描述符
mov dword [ebx+0x20],0x80007fff          ;基地址为 0xB8000，界限 0x07FFF
mov dword [ebx+0x24],0x0040920b          ;粒度为字节，数据段，DPL=00
```

注意代码中的粗体部分，对照一下段描述符的格式，你会发现，这些段描述符的DPL 都是0。也就是说，我们将这些段的特权级定为最高级别。

特权级保护机制只在保护模式下才能启用，而进入保护模式的方法是设置CR0 寄存器的PE位。而且，处理器建议，在进入保护模式后，执行的第一条指令应当是跳转或者过程调用指令，以清空流水线和乱序执行的结果，并串行化处理器，就像这样：

```
jmp dword 0x0010:flush
```

转移到的目标代码段是刚刚定义过的，描述符特权级DPL 为0。要将控制转移到这样的代码段，当前特权级CPL 必须为0。不过，这并不是问题。进入保护模式之后，处理器自动将当前特权级CPL 设定为0，以0 特权级的身份开始执行保护模式的初始指令。

参见第11 章里的图11-10，段选择子实际上由三部分组成，分别是描述符的索引号、表指示器TI 和RPL 字段。在以上指令中，段选择子0x0010 的TI 位是0，意味着目标代码段的描述符在GDT 中。该选择子索引字段的值是2，指向（GDT 中的）2 号描述符。

GDT 中的1 号描述符是保护模式下的初始代码段描述符，特权级DPL 为0，而当前特权级CPL 也是0，从初始的0 特权级转移到另一个0 特权级的代码段，这是允许的。转移之后，jmp 指令中的选择子0x0010

被加载到段寄存器CS，其低两位采用目标代码段描述符DPL 的值。也就是说，控制转移之后，当前特权级仍为0。

这里遗漏了一样东西，尽管它对于处理器的特权级检查来说很重要，但更多的时候是个累赘。那就是选择子中的RPL 字段。

RPL 的意思是请求特权级（Requested Privilege Level）。我们知道，要将控制从一个代码段转移到另一个代码段，通常是使用jmp 和call 指令，并在指令中提供目标代码段的选择子，以及段内偏移量（入口点）。而为了访问内存中的数据，也必须先将段选择子加载到段寄存器DS、ES、FS 或者GS 中。不管是实施控制转移，还是访问数据段，这都可以看成是一个请求，请求者提供一个段选择子，请求访问指定的段。从这个意义上来说，RPL 也就是指请求者的特权级别（Requestor’s Privilege Level）。

在绝大多数时候，请求者都是当前程序自己，因此，CPL＝RPL。要判断请求者是谁，最简单的方法就是看谁提供了选择子。以下是两个典型的例子：

代码清单13-1 中的第55 行：

```
jmp dword 0x0010:flush
```

在这里，提供选择子0x0008 的是当前程序自己。

再比如同一代码清单中的第59、60 行：

```
mov eax,0x0008              ;加载数据段（0～4GB）选择子
mov ds,eax
```

非常清楚的是，这同样是当前程序自己拿着段选择子0x0008 来“请求”代入段寄存器DS，以便在随后的指令中访问该段中的数据。

但是，在一些并不多见的情况下，RPL 和CPL 并不相同。如图14-6 所示，特权级为3 的应用程序希望从硬盘读一个扇区，并传送到自己的数据段，因此，数据段描述符的DPL 同样会是3。

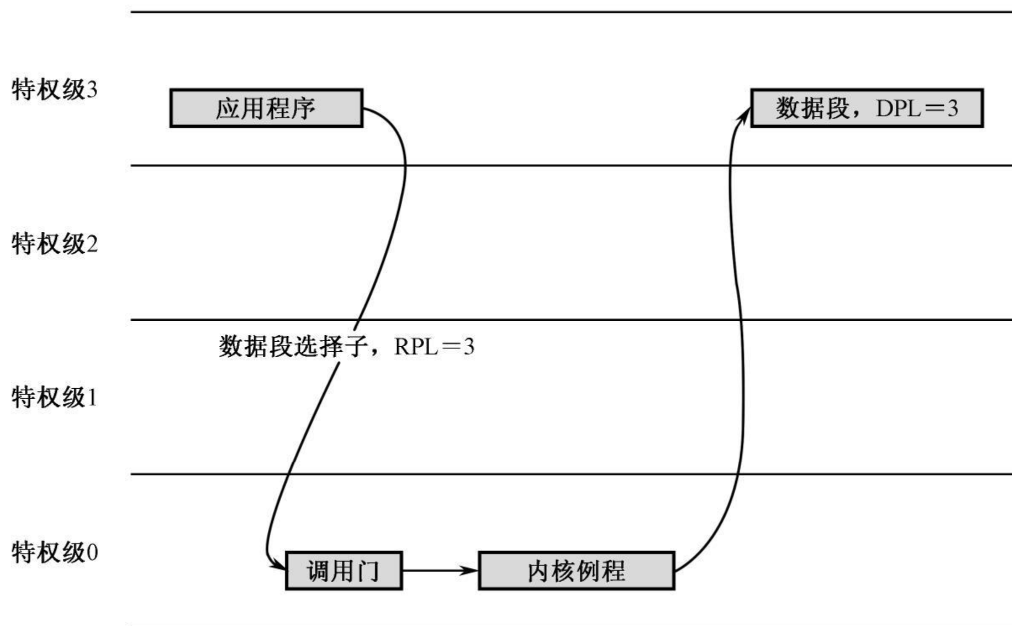


图14-6　请求特权级RPL 和当前特权级CPL 不相同的例子

由于I/O 特权级的限制，应用程序无法自己访问硬盘。好在位于0 特权级的操作系统提供了相应的例程，但必须通过调用门才能使用，因为特权级间的控制转移必须通过门。假设，通过调用门使用操作系统例程时，必须传入3 个参数，分别是CX 寄存器中的数据段选择子、EBX 寄存器中的段内偏移，以及EAX 中的逻辑扇区号。

高特权级别的程序可以访问低特权级别的数据段，这是没有问题的。因此，操作系统例程会用传入的数据段选择子代入段寄存器，以便代替应用程序访问那个段：

```
mov ds,cx
```

在执行这条指令时，CX 寄存器中的段选择子，其RPL 字段的值是3，当前特权级CPL 已经变成0，因为通过调用门实施控制转移可以改变当前特权级。显然，请求者并非当前程序，而是特权级为3 的应用程序，RPL 和CPL 并不相同。

不过，上面的例子只是表明RPL 有可能和CPL 并不相同，但并没有说明引入RPL 到底有什么必要性，它似乎是多余的，没有它，程序也能正常工作，不是吗？如果你是这样想的，那就来看看下面这个例子。

如图14-7 所示，人类的可恶之处是无孔不入，总爱钻空子。想象一下，应用程序的编写者通过钻研，知道了操作系统数据段的选择子，而且希望用这个选择子访问操作系统的数据段。当然，他不可能在应用程序里访问操作系统数据段，因为那个数据段的DPL 为0，而应用程序工作时的当前特权级为3，处理器会很机警地把来访者拒之门外。

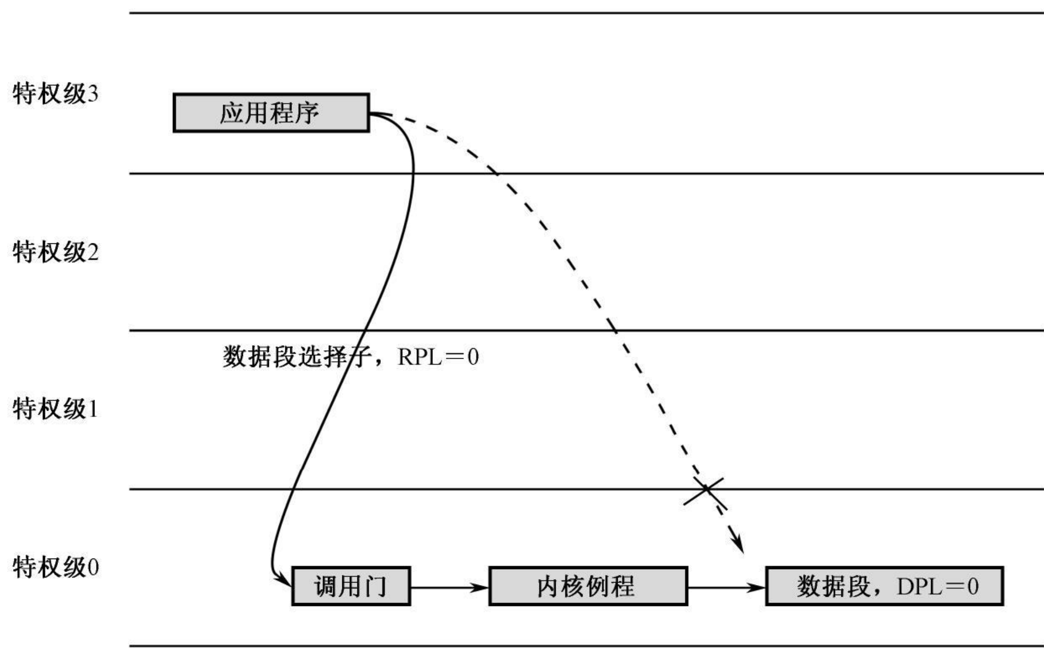


图14-7　在特权级检查中引入RPL 的必要性

但是，他可以借助于调用门。调用门工作在目标代码段的特权级上，一旦处理器的执行流离开应用程序，通过调用门进入操作系统例程时，当前特权级从3 变为0。当那个不怀好意的程序将一个指向操作系统数据段的选择子通过CX 寄存器作为参数传入调用门时，因为当前特权级已经从3 变为0，可以从硬盘读出数据，并且允许向操作系统数据段写入扇区数据，他得逞了！

处理器的智商很低，它不可能知道谁是真正的请求者。作为最聪明的灵长类动物，你当然可以通过分析程序的行为来区分它们，但处理器不能。因此，当指令

```
mov ds,ax
```

或者

```
mov ds,cx
```

执行时，AX 或者CX 寄存器中的选择子可能是操作系统自己提供的，也可能来自于恶意的用户程序，这两种情况要区别对待，但已经超出了处理器的能力和职权范围。

怎么办?

看得出来，单纯依靠处理器硬件无法解决这个难题，但它可以在原来的基础上多增加一种检查机制，并把如何能够通过这种检查的自由裁量权交给软件（的编写者）。

引入请求特权级（RPL）的原因是处理器在遇到一条将选择子传送到段寄存器的指令时，无法区分真正的请求者是谁。但是，引入RPL 本身并不能完全解决这个问题，这只是处理器和操作系统之间的一种协议，处理器负责检查请求特权级RPL，判断它是否有权访问，但前提是提供了正确的RPL；内核或者操作系统负责鉴别请求者的身份，并有义务保证RPL 的值和它的请求者身份相符，因为这是处理器无能为力的。

因此，在引入RPL 这件事上，处理器的潜台词是，仅依靠现有的CPL 和DPL，无法解决由请求者不同而带来的安全隐患。那么，好吧，再增加一道门卫，但前提是，操作系统只将通行证发放给正确的人。

操作系统的编写者很清楚段选择子的来源，即，真正的请求者是谁。当它自己读写一个段时，这没有什么好说的；当它提供一个服务例程时，3 特权级别的用户程序给出的选择子在哪里，也是由它定的，它也知道。在这种情况下，它所要做的，就是将该选择子的RPL 字段设置为请求者的特权级（可以使用arpl 指令，将在本章的后面介绍）。剩下的工作就看处理器了。每当处理器执行一个将段选择子传送到段寄存器（DS、ES、FS、GS）的指令，比如：

```
        mov ds,cx
```

时，会检查以下两个条件是否都能满足。

● 当前特权级CPL 高于或者和数据段描述符的DPL 相同。即，在数值上，CPL≤数据段描述符的DPL；

●请求特权级RPL 高于或者和数据段描述符的DPL 相同。即，在数值上，RPL≤数据段描述符的DPL。

如果以上两个条件不能同时成立，处理器就会阻止这种操作，并引发异常中断。

按照Intel 公司的说法，引入RPL 的意图是“确保特权代码不会代替应用程序访问一个段，除非应用程序自己拥有访问那个段的权限”。多数读者都只在字面上理解这句话的意思，而没有意识到，这句话只是如实地描述了处理器自己的工作，并没有保证它可以鉴别RPL 的有效性。

最后，我们来总结一下基本的特权级检查规则。

首先，将控制直接转移到非依从的代码段，要求当前特权级CPL 和请求特权级RPL 都等于目标代码段描述符的DPL。即，在数值上，

```
CPL＝目标代码段描述符的 DPL
RPL＝目标代码段描述符的 DPL
```

一个典型的例子就是使用jmp 指令进行控制转移：

```
jmp 0x0012:0x00002000
```

因为两个代码段的特权级相同，故，转移后当前特权级不变。

其次，要将控制直接转移到依从的代码段，要求当前特权级CPL 和请求特权级RPL 都低于，或者和目标代码段描述符的DPL 相同。即，在数值上，

```
CPL≥目标代码段描述符的 DPL
RPL≥目标代码段描述符的 DPL
```

控制转移后，当前特权级保持不变。

通过门实施的控制转移，其特权级检查规则将在相应的章节里详述。

第三，高特权级别的程序可以访问低特权级别的数据段，但低特权级别的程序不能访问高特权级别的数据段。访问数据段之前，肯定要对段寄存器DS、ES、FS 和GS 进行修改，比如

```
mov fs,ax
```

在这个时候，要求当前特权级CPL 和请求特权级RPL 都必须高于，或者和目标数据段描述符的DPL 相同。即，在数值上，

```
CPL≤目标数据段描述符的 DPL
RPL≤目标数据段描述符的 DPL
```

最后，处理器要求，在任何时候，栈段的特权级别必须和当前特权级CPL 相同。因此，随着程序的执行，要对段寄存器SS 的内容进行修改时，必须进行特权级检查。以下就是一个修改段寄存器SS 的例子：

```
mov ss,ax
```

在对段寄存器SS 进行修改时，要求当前特权级CPL 和请求特权级RPL 必须等于目标栈段描述符的DPL。即，在数值上，

```
CPL＝目标栈段描述符的 DPL
RPL＝目标栈段描述符的 DPL
```

0 特权级是最高的特权级别，当一个系统的各个部分都位于0 特权级时，各种特权级检查总能够获得通过，就像这种检查和检验并不存在一样。所以，处理器的设计者建议，如果不需要使用特权机制的话，可以将所有程序的特权级别都设置为0，就像我们一直所做的那样。

小结

1．程序员在写程序时，不需要指定特权级别。当程序运行时，操作系统将程序创建为任务局部空间的内容，并赋予较低特权级别，比如3，操作系统对应着任务全局空间的内容。如果有多个任务，则操作系统属于所有任务的公共部分。

2．当任务运行在局部空间时，可以在各个段之间转移控制，并访问私有数据，因为它们具有相同的特权级别，但不允许直接将控制转移到高特权级别的全局空间的段，除非通过调用门，或者目标段是依从的代码段。

3．当通过调用门进入全局空间执行时，操作系统可以在全局空间内的各个段之间转移控制并访问数据，因为它们也具有相同的特权级别。同时，操作系统还可以访问任务局部空间的数据，即低特权级别的数据段。但除了调用门返回外，不允许将控制转移到低特权级别的局部空间内的代码段。

4．任何时候，当前栈的特权级别必须和CPL 是一样的。进入不同特权级别的段执行时，要切换栈，这是以后要讲述的内容。

检测点14.1

1． 选择填空：x86 处理器提供了4 个特权级别0、1、2 和3。较小的数字拥有较（ ）的特权级别，其中3 特权级是最（ ）的低权级别。可选择答案：A.低 B.高

2． 将控制转移到另一个代码段时，如果目标段不是依从的，并且转移时不通过门，则CPL、RPL 和DPL 之间的关系必须符合______________________的条件；如果目标段是依从的，则必须符合______________________的条件。

3． 如果当前特权级别CPL 为2，那么，它可以访问DPL 为_______的数据段。

14.2　代码清单14-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：14-1（保护模式微型核心程序）

源程序文件：c14_core.asm

14.3　内核程序的初始化

本章没有提供主引导程序，因为我们要继续使用上一章的主引导程序。毕竟，主引导程序只是用来加载内核程序，并执行前期的内核初始化工作。主引导程序工作在0 特权级。

现在，让我们来分析本章代码清单14-1，这是前一章内核程序的修改版本，使用了任务、LDT、TSS 和特权级等最新的处理器特性和工作机制。代码清单中，一开始的常数定义以及程序头部的格式和前一章也完全相同，这是可以理解的，作为主引导程序和内核程序的协议部分，它们总应该是稳定不变的。

文件起始部分的常数定义了内核所有段的选择子。很显然，这些选择子的RPL 字段都是0，内核请求访问自己的段，请求特权级应当为0。

内核的入口点在第775 行。在执行到这里的时候，主引导程序已经加载了内核，并对它进行了前期的初始化工作。

因为加载的是内核程序，而内核应当工作在0 特权级，所以主引导程序在初始化内核时，所创建的描述符，其目标特权级DPL 都为0，如图14-8 所示。注意，这些描述符都是在GDT中创建的，图中左边是各描述符在GDT 中的偏移量，右边是各个描述符的选择子。

GDT内偏移		描述符索引号
+38	核心代码段（位置和长度不定，DPL＝0）	38
+30	核心数据段（位置和长度不定，DPL＝0）	30
+28	公用例程段（00040000～长度不定，DPL＝0）	28
+20	文本模式显存（000B8000～000BFFFF，DPL＝0）	20
+18	初始栈段（00006C00～00007C00，DPL＝0）	18
+10	初始代码段（00007C00～00007DFF，DPL＝0）	10
+08	0～4GB数据段（00000000～FFFFFFFF，DPL＝0）	08
+00	空描述符	00

图14-8　内核加载完成后的GDT 布局

这些描述符所指向的段，有的是代码段，有的是数据段。如果是数据段，则只有内核自己才能访问，因为其描述符的DPL 是0，低特权级别的程序访问这些段时，会被阻止以防出现安全问题；如果是代码段，则通常只有0 特权级的程序才能将控制转移到该段，也就是说，只能从内核其他正在执行的部分转移到该段执行，因为它们的特权级别相同。

第779～809 行，用于在屏幕上显示初始的信息，包括一个欢迎信息和一个处理器品牌信息。

14.3.1　调用门

在上一章里，内核的主要功能是加载和重定位用户程序，并将处理器的控制权移交过去。用户程序执行完毕，还要重新回收控制。现在我们已经知道，在上一章里，内核赋予用户程序的特权级别是0，所以用户程序是在0 特权级上运行的。也正是因为如此，当用户程序通过U-SALT 表中的符号地址直接调用内核例程时，才会通过特权级检查。

在本章里，内核也做同样的工作。不同之处在于，它将用户程序的特权级定为3，也就是最低的特权级别。没有人愿意将自己的程序放在特权级3 上，但系统核心一定会将它放在特权级3 上。

尽管保护模式非常复杂，但这并没有加重用户程序（应用程序）编写者（程序员）的负担，因为他们不必考虑底层的很多东西，这也是为什么本章没有提供用户程序代码清单的原因。事实上，本章将继续沿用第13 章的用户程序，只不过要作为一个任务进行加载，加载的方法和上一章是不同的。而且，运行时的特权级别是3，不再是上一章中的0。

为了方便应用程序的编写，内核通常要提供大量的例程供它们调用。例如，在第13 章中，用户程序可以调用内核例程@PrintString 和@ReadDiskData。为此，用户程序需要定义SALT 表，并在表中填写例程的符号名。之后，再由内核将符号名转换成入口地址，也就是该例程所对应的段选择子和段内偏移量。

例程是由内核提供的，它们的特权级通常就是内核的特权级。在上一章里，内核程序和用户程序都运行在0 特权级，而且都是普通的段间控制转移，所以，在用户程序内直接调用内核例程，这不会有任何问题。

但是，考虑一下，在本章中，用户程序运行时的特权级别将会是3。由于处理器禁止将控制从特权级低的程序转移到特权级高的程序，因此，如果还像以前那样直接调用内核例程，百分之百不会成功，一定会引发处理器异常中断。但是，现实的需求也不能不予考虑，任何操作系统都应当提供大量的功能调用服务。为此，需要安装调用门。

调用门（Call-Gate）用于在不同特权级的程序之间进行控制转移。本质上，它只是一个描述符，一个不同于代码段和数据段的描述符，可以安装在GDT 或者LDT 中。该描述符的格式如图14-9 所示，下面是低32 位，上面是高32 位。

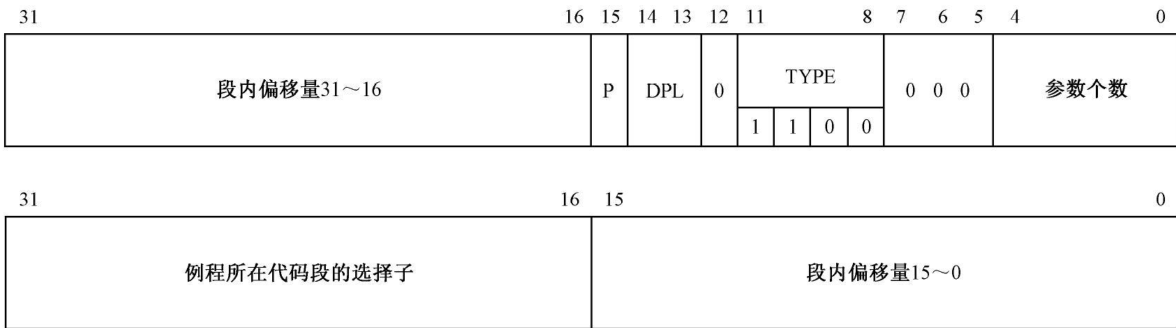


图14-9 调用门描述符的格式

如图15-9 所示，调用门描述符给出了例程所在代码段的选择子，而不是32 位线性地址。有了段选择子，就能访问描述符表得到代码段的基

地址，这样做无非是间接了一点，但却可以在通过调用门进行控制转移时，实施代码段描述符有效性、段界限和特权级的检查。

例程在代码段中的偏移量也是在描述符中直接指定的，只是被分成了两个16 位的部分。很显然，在通过调用门调用例程时，不使用指令中给出的偏移量。

描述符中的TYPE 字段用于标识门的类型，共4 比特，值“1100”表示调用门。

描述符中的P 位是有效位，通常应该是“1”。当它为“0”时，调用这样的门会导致处理器产生异常中断。对于操作系统来说，这个机关可能会很有用。比如，为了统计调用门的使用频率，可以将它置“0”。然后，每当因调用该门而产生异常中断时，在中断处理程序中将该门的调用次数加一，同时把P 位置“1”。对于因P 位为“0”而引起的中断来说，它们属于故障中断，从中断处理过程返回时，处理器还会重新执行引起故障的指令。此时，因P 已经为“1”，所以可以执行。就当前的例子而言，因为在提供调用门服务的同时，还要统计门的调用次数，故，可以在该调用门所对应的例程中将P 位清零。这样，下一次该门被调用时，又会重复以上过程。

通过调用门实施特权级之间的控制转移时，可以使用jmp far 指令，也可以使用call far 指令。如果是后者，会改变当前特权级CPL。因为栈段的特权级必须同当前特权级保持一致，因此，还要切换栈，即，从低特权级的栈切换到高特权级的栈。比如，一个特权级为3 的程序必须使用自己的3 特权级栈工作。当它通过调用门进入0 特权级的代码段执行时，当前特权级由3 变为0。此时，栈也要跟着切换，从3 特权级的栈切换到0 特权级的栈。这主要是为了防止因栈空间不足而产生不可预料的问题，同时也是为了防止栈数据的交叉引用。

为了切换栈，每个任务除了自己固有的栈之外，还必须额外定义几套栈，具体数量取决于任务的特权级别。0 特权级任务不需要额外的栈，它自己固有的栈就足够使用，因为除了调用返回外，不可能将控制转移到低特权级的段；1 特权级的任务需要额外定义一个描述符特权级DPL 为0 的栈，以便将控制转移到0 特权级时使用；2 特权级的任务则需要额外定义两个栈，描述符特权级DPL 分别是0 和1，在控制转移到0 特权级和1 特权级时使用；3 特权级的任务最多额外定义3 个栈，描述符特权级分别是0、1 和2，在控制转移到0、1 和2 特权级时使用。

不要担心，这些额外的栈，也会由操作系统加载程序时自动创建，本章的源代码就演示了这一过程。想想看，如果这一切都由你来做，你一定不会把自己程序的特权级别定得很低，以致于还要切换栈段，对不对？

这些额外创建的栈，其描述符位于任务自己的LDT 中。同时，还要在任务的TSS 中登记，原因是，栈切换是由处理器固件自动完成的，处理器需要根据TSS 中的信息来完成这一过程。如图14-2 所示，在TSS 内，从偏移4～24 处登记有特权级0 到2 的栈段选择子，以及相应的ESP 初始值。任务自己固有的栈信息则位于偏移量为56（ESP）和80（SS）的地方。

任务寄存器TR 总是指向当前任务的任务状态段TSS，其内容为该TSS 的基地址和界限。在切换栈时，处理器可以用TR 找到当前任务的TSS，并从TSS 中获取新栈的信息。

通过调用门使用高特权级的例程服务时，调用者会传递一些参数给例程。如果是通过寄存器传送，这没有什么可说的。不过，要传递的参数很多时，更经常的做法是通过栈进行。调用者把参数压入栈，例程从栈中取出参数。在高级语言里，这是一贯的做法。

例程需要什么参数，先压入哪个参数，后压入哪个参数，这是调用者和例程之间的约定，调用者是清楚的。否则，它不会调用这个例程。但是，这一切对于处理器来说是懵懂的。特别是，当栈切换时，参数还在旧栈中。为了使例程能获得参数，必须将参数从旧栈复制到新栈中。

参数的复制工作是由处理器固件完成的，但它必须事先知道参数的个数，并根据该数量决定复制多少内容。所以，调用门描述符中还有一个参数个数字段，共5 比特。就是说，至多允许传送31 个参数。

栈切换前，段寄存器SS 指向的是旧栈，ESP 指向旧栈的栈顶，即最后一个被压入的过程参数；栈切换后，处理器自动替换SS 和ESP 寄存器的内容，使它们分别为新栈的选择子和新栈的栈顶（最后一个被复制的参数）。这一切，对程序的编写者来说是透明的。所谓“透明”，就是说，程序员不用关心栈的切换和参数的复制，他即使不知道还有栈切换这回事，也不会影响程序编写工作。因为，在栈切换前，

```
        pop edx
```

可以得到最后一个被压入的参数，在栈切换后，这条指令同样可以得到那个参数，尽管栈段和栈顶指针已经改变。

调用门描述符中的DPL 和目标代码段描述符的DPL 用于决定哪些特权级的程序可以访问此门。具体的规则是必须同时符合以下两个条件才行：

●当前特权级CPL 和请求特权级RPL 高于，或者和调用门描述符特权级DPL 相同。即，在数值上

```
CPL≤调用门描述符的 DPL
RPL≤调用门描述符的 DPL
```

●当前特权级CPL 低于，或者和目标代码段描述符特权级DPL 相同。即，在数值上

```
CPL≥目标代码段描述符的 DPL
```

举个例子，如果调用门描述符的DPL 为2，那么，只有特权级为0、1 和2 的程序才允许使用该调用门，特权级为3 的程序使用此门将引发处理器异常中断。

如图14-10 所示，调用门的DPL 是特权级检查的下限。除此之外，目标代码段的特权级也是需要考虑的因素。调用门描述符中有目标代码段的选择子，它指向目标代码段的描述符。当一个程序通过调用门转移控制时，处理器还要检查目标代码段描述符的DPL，该DPL 决定了调用门特权级检查的上限。也就是说，只有那些特权级低于或者等于目标代码段DPL 的程序才允许使用此门。

调用门描述符中有一些字段没有使用，固定为“0”。

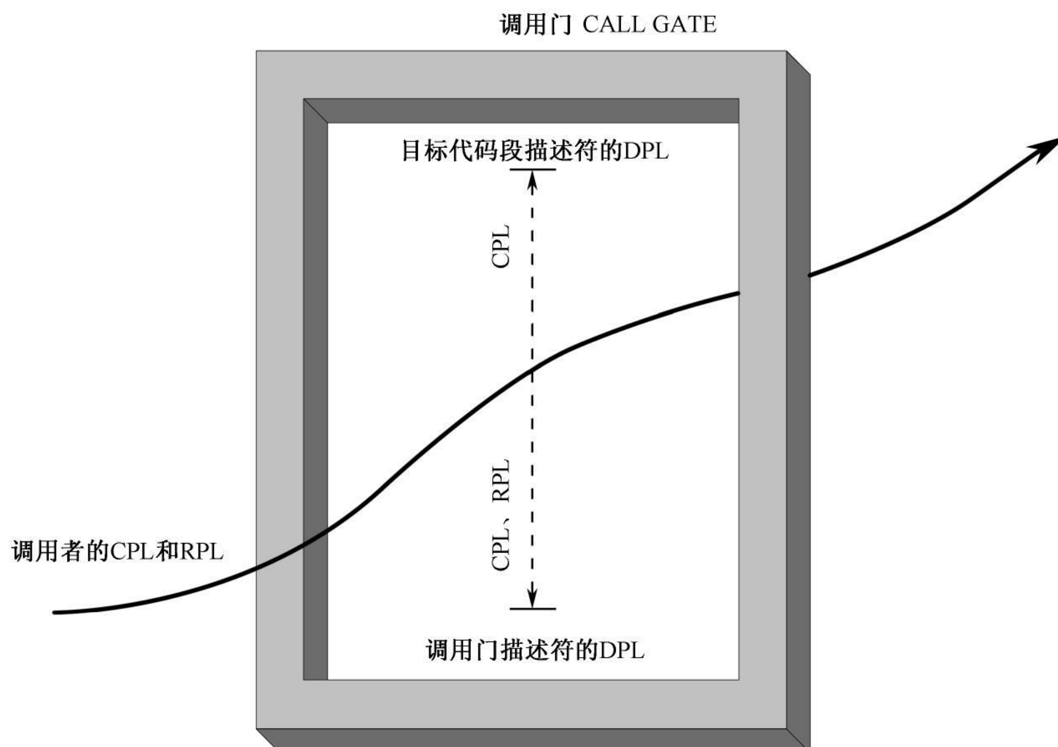


图14-10　调用门的基本特权级检查规则

14.3.2　调用门的安装和测试

第812～826 行用于安装调用门的描述符，其实也就等于是在安装调用门。

安装的调用门供其他特权级的程序使用，它们在本质上是一些例程，这些例程在上一章里使用过，相信都不会陌生。在上一章里，所有对外公开的例程都以字符串的形式定义在SALT 表中，该表位于内核数据段。

内核数据段中的SALT 表简称C-SALT，位于代码清单14-1 的第364～383 行，属于内核数据段。该表由多个条目组成，每个条目262 字节，其中，前256 字节是例程的名字，后6 字节是例程的地址（前4 字节是例程在目标代码段内的偏移量，后2 字节是例程所在代码段的选择子）。

所有例程都位于公共例程段中，而公共例程段的DPL 是0。为了使其他特权级的程序也能使用这些例程，必须将C-SALT 表中的例程地址转换成调用门。

转换过程使用了循环。转换时需要定位每一个条目，故，第812 行用于将C-SALT 表的起始偏移地址传送到EDI 寄存器，这是第一个条目的位置，以后每次加上262，就能对准下一个条目。

循环次数是由条目数量控制的，条目数是常数salt_items，位于第386 行，第813 行的指令用于将它作为立即数传送到ECX 寄存器。

循环的结构是这样的：

```
814    .b3:
815        push ecx

           ;这里是具体执行转换的指令

824        add edi,salt_item_len     ;指向下一个 C-SALT 条目
825        pop ecx

826        loop .b3
```

因为在转换过程中要用到ECX 寄存器，所以在每次循环的一开始，要先压栈保存ECX 寄存器的值，然后，在loop 指令执行前恢复。

在循环体内，第816 行，用于将每个条目（例程）的32 位段内偏移地址传送到EAX 寄存器。每个条目的长度是262 字节，而它的偏移地址则位于256 字节处。第817 行，用于获取条目（例程）所在代码段的选择子，它位于条目内第260 字节处。

创建调用门描述符的工作实际上是调用过程make_gate_descriptor 来完成的。该过程位于第331 行，属于公共例程段。调用该过程时，需要传入三个参数，分别是EAX 寄存器中的32 位偏移地址、BX 寄存器中的代码段选择子，以及CX 寄存器中的门属性。调用门的属性字段是2 字节的长度，通过CX 寄存器传入门属性时，必须保证各属性位都在原始位置。在我们的代码中，每次通过CX 寄存器传入的值是

```
818        mov cx,1_11_0_1100_000_00000B
```

很显然，P＝1，DPL＝3，即，只有特权级高于等于3 的代码段才能调用此门，参数的数量为0，也就是不需要通过栈传递参数。

下面我们来看一看例程make_gate_descriptor 都做了些什么。

第340～342 行，先在EDX 寄存器中得到32 位偏移地址的复制品，然后将低16 位清除，只留下32 位偏移地址的高16 位部分，并同CX 寄存器中的属性值一起，形成调用门描述符的高32 位。

第344～346 行，将EAX 寄存器的高16 位清除，只留下32 位偏移地址的低16 位。接着，将EBX 寄存器逻辑左移16 次，使得段选择子位于它的高16 位。最后，用or 指令将这两个寄存器合并，就得到了调用门描述符的低32 位。

第351 行，retf 指令使得控制返回调用者。注意，从这条指令可以看出，该过程必须以远调用的方式使用。

回到内核代码段。

第821 、822 行， 在调用了例程make_gate_descriptor 后， 立即调用了另一个例程set_up_gdt_descriptor 来安装刚才创建的调用门描述符。在GDT 中安装描述符的过程和前一章相同，不再讲述。显然，调用门描述符是在GDT 中创建的，并用CX 寄存器返回该描述符的选择子，即调用门选择子。

第823 行，将返回的调用门选择子回填到条目内，用以覆盖原先的代码段选择子。

取决于C-SALT 表的大小，循环过程会进行多次。在本章中，C-SALT 表中共有4 个条目，这4 个调用门安装之后，GDT 的布局如图14-11 所示。

第829、830 行对刚安装好的调用门进行测试，看它好不好用。测试的结果是在屏幕上显示一行文字，意思为“系统范围内的调用门已经安装”。

标号salt_1 指向C-SALT 表中第一个条目的起始处，在此基础上增加256，就是它的地址部分。现在我们已经知道，该条目对应着公共例程段中的put_string 过程，用于显示零终止的字符串。

表面上，这是一条普通的间接绝对远调用指令call far，通过指令中给出的地址操作数，可以间接取得32 位的偏移地址和16 位的代码段选择子，这样的指令我们太熟悉了。但是，处理器在执行这条指令时，会用该选择子访问GDT/LDT，检查那个选择子，看它指向的是调用门描述符，还是普通的代码段描述符。如果是前者，就按调用门来处理；如果是后者，还按一般的段间控制转移处理。

在这里，因为salt_1 条目的选择子已经被替换成调用门选择子，所以处理器按调用门的方式来执行控制转移。通过调用门实施控制转移时，处理器只用选择子部分，salt_1 条目中给出的32位偏移量部分被丢弃。原因很简单，通过调用门进行控制转移不需要偏移量，偏移量已经在调用门描述符中给出了。不单单是间接绝对远调用，直接绝对远调用也是这样，如果选择子指向的是调用门，偏移量也会被忽略，例如：

```
        call 0x0040:0x0000c000
```

在这个例子中，因为是通过调用门实施控制转移，处理器将忽略偏移量0x0000c000。

GDT内偏移		描述符索引号
+58	调用门（@TerminateProgram，DPL＝3）	58
+50	调用门（@PrintDwordAsHexString，DPL＝3）	50
+48	调用门（@ReadDiskData，DPL＝3）	48
+40	调用门（@RrintString，DPL＝3）	40
+38	核心代码段（位置和长度不定，DPL＝0）	38
+30	核心数据段（位置和长度不定，DPL＝0）	30
+28	公用例程段（00040000～长度不定，DPL＝0）	28
+20	文本模式显存（000B8000～000BFFFF，DPL＝0）	20
+18	初始栈段（00006C00～00007C00，DPL＝0）	18
+10	初始代码段（00007C00～00007DFF，DPL＝0）	10
+08	0～4GB数据段（00000000～FFFFFFFF，DPL＝0）	08
+00	空描述符	00

图14-11　安装调用门后的GDT 布局

借助调用门，当程序的执行流从低特权级的代码段转入高特权级的代码段时，如果那是个非依从的代码段，当前特权级也随之变为目标代

码段的特权级。不过，如果调用者和被调用者的特权级相同，则特权级不会变化。在当前的例子中，是从内核代码段调用公共例程段的例程，尽管也是通过调用门，但它们的特权级都是0。所以，在控制转移的过程中不会发生栈切换，仅仅是把返回地址CS 和EIP 压入当前栈。当执行retf 指令后，处理器从栈中恢复CS 和EIP 的原始内容，于是又返回到原先的代码段接着执行。

事实上，能够通过调用门发起控制转移的指令还包括jmp，但只用在不需要从调用门返回的场合下，而且不改变当前特权级。也就是说，目标代码是在当前特权级上执行。

通过调用门进行控制转移的特权级检查，既要在转移前进行，而且，还要在控制返回时进行。完整的特权级检查过程将在本章的后面进一步说明。

检测点14.2

1． 通过调用门转移控制时，CPL、RPL 和目标代码段描述符的DPL 必须在数值上符合____________ 的条件； CPL 、RPL 和调用门描述符的DPL 必须在数值上符合____________的条件。即，只能通过调用门将控制转移到与当前特权级相同或者更高的代码段。

2． 调用门描述符只能安装在GDT 中吗？如果某调用门描述符的值是0x0000CC0000552FC0，那么，目标代码段的选择子是_____，段内偏移量为_____，描述符的特权级是_____，目标代码段的特权级是_____，要通过此门转移控制，CPL和RPL要符合什么条件才行？

14.4 加载用户程序并创建任务

14.4.1 任务控制块和TCB 链

继续讲解代码清单14-1。

第832、833 行是以传统的方式调用内核例程显示字符串。即使不通过调用门，特权检查也是照常进行的，而且更为严格。把控制从较低的特权级转移到较高的特权级，通过调用门尚有可能，但直接控制转移则在任何时候都是不允许的。当然，在这里，是从0 特权级的内核代码段进入同样是0 特权级的公共例程段，能够通过特权级检查。

在内核初始化完成后，和第13 章一样，接下来的工作就是加载和重定位用户程序（ 应用程序），并移交控制权。按处理器的要求标准，要使一个程序成为“任务”，并且能够参与任务切换和调度，那不是简简单单就能行的，必须要有LDT 和TSS。而为了创建这两样东西，又需要更多的东西。所以，加载和执行用户程序的活儿，比起从前是麻烦了不少。不信？一会儿就要做这件事，到时候你就知道了。

加载程序并创建一个任务，需要用到很多数据，比如程序的大小、加载的位置，等等。当任务执行结束，还要依据这些信息来回收它所占用的内存空间（在本书中没有体现，但一个合格的操作系统必须实现该功能）。还有，多任务系统是多个任务同时运行的，特别是在一个单处理器（核）的系统中，为了在任务之间切换和轮转，必须能追踪到所有正在运行的任务，记录它们的状态，或者根据它们的当前状态来采取适当的操作（在本书的第16 章，将学习任务的切换和轮转技术）。

为了满足以上的要求，内核应当为每一个任务创建一个内存区域，来记录任务的信息和状态，称为任务控制块（Task Control Block，TCB）。任务控制块不是处理器的要求，是我们自己为了方便而发明的。如图14-12 所示，这是任务控制块的结构，很明显，这里有两种大小的方格，较窄的格子代表16位的数据宽度，即1 个字；而较宽的格式代表32 位的数据宽度，即2 个字。注意，不要纠结于表中的内容和细节，有个大概印象即可。

偏移	15 ～ 0
+0x46 ->	头部选择子
+0x44 ->	2特权级栈的初始ESP
+0x40 ->	2特权级栈选择子
+0x3E ->	2特权级栈基地址
+0x3A ->	2特权级栈以4KB为单位的长度
+0x36 ->	1特权级栈的初始ESP
+0x32 ->	1特权级栈选择子
+0x30 ->	1特权级栈基地址
+0x2C ->	1特权级栈以4KB为单位的长度
+0x28 ->	0特权级栈的初始ESP
+0x24 ->	0特权级栈选择子
+0x22 ->	0特权级栈基地址
+0x1E ->	0特权级栈以4KB为单位的长度
+0x1A ->	TSS选择子
+0x18 ->	TSS基地址
+0x14 ->	TSS界限值
+0x12 ->	LDT选择子
+0x10 ->	LDT基地址
+0x0C ->	LDT当前界限值
+0x0A ->	程序加载基地址
+0x06 ->	任务状态
+0x04 ->	下一个TCB基地
+0x00 ->	

图14-12　任务控制块TCB 的结构

为了能够追踪到所有任务，应当把每个任务控制块TCB 串起来，形成一个链表（链表是一种数据结构，有一门计算机课程就叫做《数据结构》）。

代码清单14-1 的第414 行，声明了标号tcb_chain 并初始化为一个双字，初始的数值为零。实际上，它是一个指针，用来指向第一个任务的TCB 线性基地址。当它为零时，表示任务的数量为0，也就是没有任务。在创建了第一个任务后，应当把该任务的TCB 线性基地址填写到这里。

每个TCB 的第一个双字，也是一个双字长度的指针，用于指向下一个任务的TCB。如果该位置是零，表示后面没有任务，这是链上的最后一个任务；否则，它的数值就是下一个任务的TCB线性基地址。如图14-13 所示，所有任务都按照被创建的先后顺序链接在一起，从tcb_chain 开始，可以依次找到每一个任务。

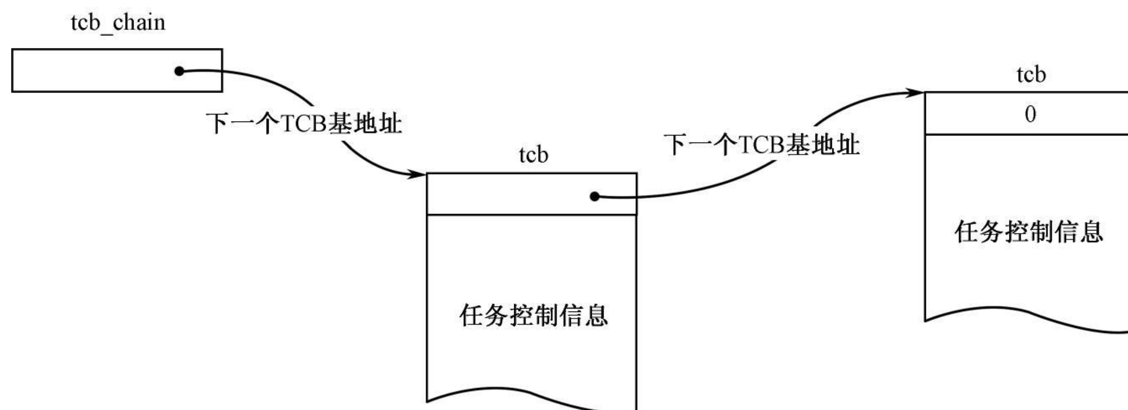


图14-13　任务控制块链

第836～838 行，用于分配创建TCB 所需要的内存空间，并将其挂在TCB 链上。如图14-12所示，当前版本的TCB 结构需要0x46 字节的内存空间。

将新TCB 追加到链表上的工作是由过程append_to_tcb_link 来完成的，位于代码清单14-1 的第735～772 行，属于内核代码段的内部（近）过程，图14-14 是它的整个流程图。

过程append_to_tcb_link 的工作思路是遍历整个链表，找到最后一个TCB，在它的TCB 指针域里填写新TCB 的首地址。它需要用ECX 作为传入的参数，ECX 的内容应当为新TCB 的线性地址。

这里有一个小小的麻烦。链首指针tcb_chain 是在内核数据段声明并初始化的，只能知道它在段内的偏移，而不知道它的线性地址，因此，只能通过内核数据段访问，而无法通过线性地址来访问；相反地，链上的每个TCB，其空间都是动态分配的，只能通过线性地址来访问。

因此，在将两个段寄存器和两个通用寄存器压栈保护之后，第742～745 行，我们令段寄存器DS 指向内核数据段以读写链首指针tcb_chain，而ES 指向整个4GB 内存空间，用于遍历和访问每一个TCB。

第747 行，要追加的TCB 一定是链表上最后一个TCB，故其用于指向下一个TCB 的指针域必须清零，以表明自己是链上最后一个TCB。每个TCB 的空间都是动态分配的，其首地址都是线性地址，只能用由段寄存器ES 所指向的4GB 段来访问。

第750～752 行，观察链首指针tcb_chain 是否为零。若为零，则表明整个链表为空，直接转移到第763 行的标号.notcb 处，在那里，直接

将链首指针指向新的TCB，恢复现场后直接返回调用者。

第754～758 行，若链首指针不为零，表明链表非空，需要顺着整个链找到最后一个TCB。和链首指针tcb_chain 不同，每个TCB 需要用4GB的段来访问，即使用段寄存器ES。

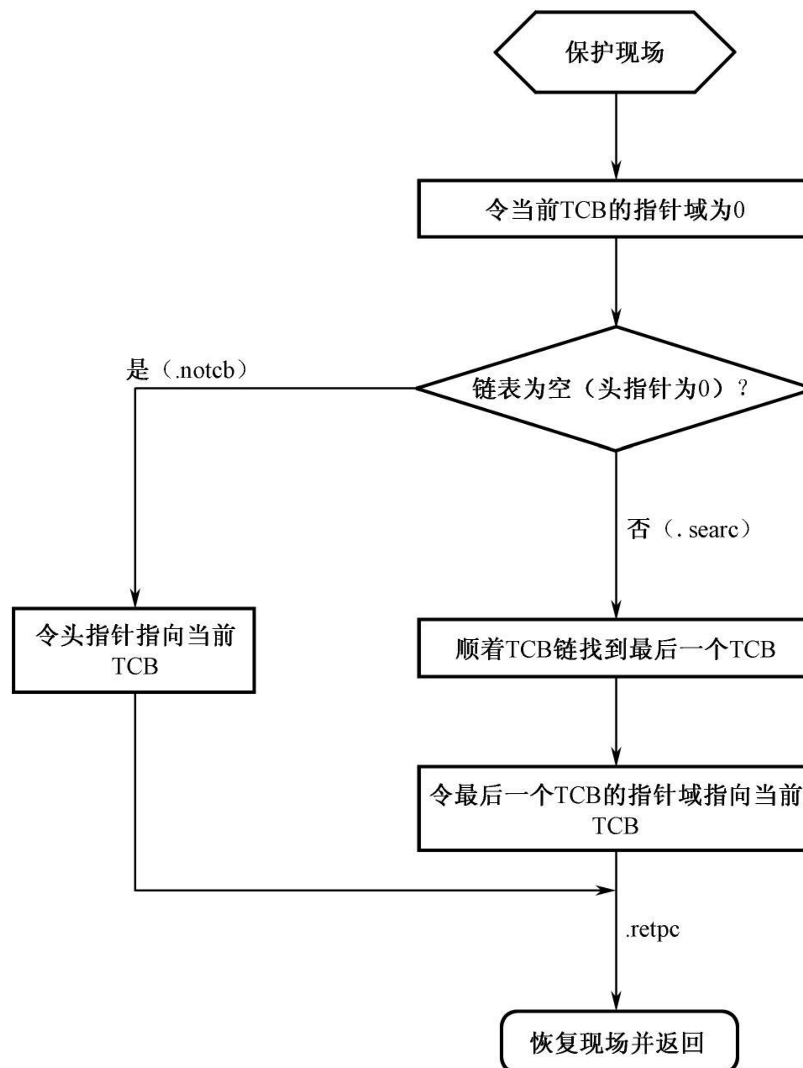


图14-14　向TCB 链上追加任务控制块的流程图

首先，将链表中要访问的那个TCB 的线性地址传送到EDX 寄存器；然后，访问它的TCB 指针域，看它是否为零。如果不为零，表明它不是链中最后一个TCB，后面还有其他TCB，于是将控制转移到.searc，令EDX 寄存器指向下一个TCB，继续搜寻。

若为零，表明它就是链上最后一个TCB，第760 行，用ECX 的内容填写其TCB 指针域，让它指向新的TCB。完成后，第761 行，直接转移

到标号.retpc 处，恢复现场并返回调用者。

14.4.2　使用栈传递过程参数

下面的工作是加载和重定位用户程序，依然是在过程load_relocate_program 中进行。该过程需要传入两个参数，分别是用户程序的起始逻辑扇区号，以及它的任务控制块TCB 线性地址。和上一章不同的是，参数不是用寄存器传入的，而是采用栈。事实上，这是更为流行和标准的做法。原因很简单，寄存器数量有限，况且还要在过程内部使用，当传入的参数很多时，栈是最好的选择。

第840～843 行，先以双字的长度将立即数50 压入当前栈，这是用户程序的起始逻辑扇区号。在第10 章里，我们已经知道push 指令可以压入立即数。因此，在这里，压入到栈中的内容将是双字0x00000032（十进制数50）。接着，再压入当前任务控制块TCB 的32 位线性地址。最后，进入过程load_relocate_program 内部执行。该过程位于第464 行，是（当前）内核代码段的内部过程。

第468～473 行，先做一些保护现场的工作，然后将栈指针寄存器ESP 的内容复制到EBP 寄存器，以访问栈中的参数。栈的访问有两种，一种是隐式的，由处理器在执行诸如push、pop、call、ret 等指令时自动进行。隐式地访问栈需要使用指令指针寄存器ESP。另一种访问栈的方式不依赖于先进后出机制，而是把栈看成是一般的数据段，直接访问其中的任何内容。在这种方式下，需要使用栈基址寄存器EBP。这里有个例子，比如，从栈中读取一个双字，该数据在栈中的偏移量是由EBP 寄存器指向的：

```
        mov edx,[ebp]
```

在32 位模式下，处理器执行这条指令时，用段寄存器SS 描述符高速缓存器中的32 位基地址，加上EBP 寄存器提供的32 位偏移量，形成32 位线性地址，访问内存取得一个双字，传送到EDX 寄存器。很显然，用EBP 寄存器来寻址时，不需要使用段超越前缀“SS:”，因为EBP 寄存器出现在指令中的地址部分时，默认使用段寄存器SS。

如图14-15 所示，这是用ESP 寄存器的内容初始化EBP 后，栈的状态。

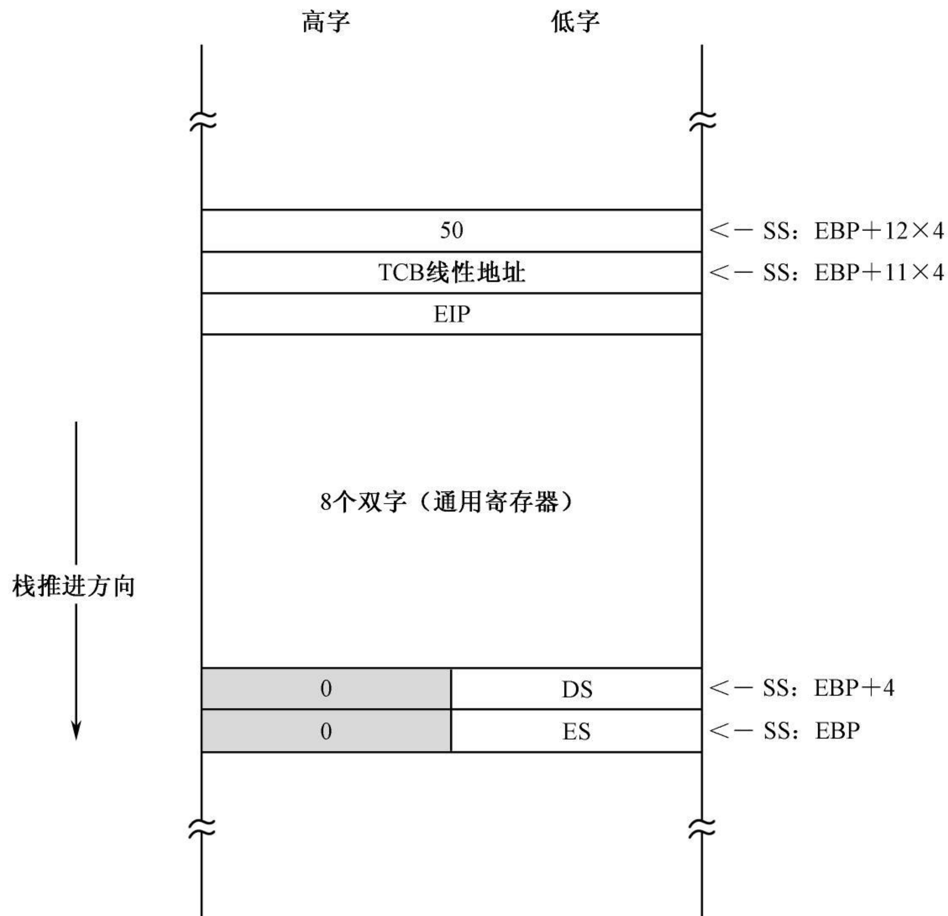


图14-15　执行mov ebp,esp 指令后的栈状态

当前的栈顶位置是SS:EBP，指向一个双字，是段寄存器ES 的内容，因为最近一次的压栈操作是

```
push es
```

在32 位模式下，访问栈用的是栈指针寄存器ESP，而且，每次栈操作的默认操作数大小是双字。处理器在执行压栈指令时，如果发现指令的操作数是段寄存器（CS、SS、DS、ES、FS、GS），那么，将先执行一个内部的零扩展操作，将段寄存器中的16 位值扩展成32 位，高16 位是全零，然后再执行压栈操作。当然，出栈指令pop 会执行相反的操作，将32 位的值截短为16位，并传送到相应的段寄存器。

相应地，SS：EBP＋4 的位置是段寄存器DS 的压栈值。因为栈是向下推进的，故较早压入的内容反而位于高地址方向，回溯它们需要增加EBP 的值。

从SS：EBP＋8 的位置开始，是pushad 指令压入的8 个双字，其中就包括EBP 在压栈时的原始内容。

再往上，是调用者的返回地址。因为load_relocate_program 是一个内部过程，是用32 位相对近调用（第843 行）进入的，故只压入了EIP 的内容，而没有压入段寄存器CS 的内容。

好了，现在终于到了我们感兴趣的地方。当初调用load_relocate_program 过程的时候，压入了两个参数，分别是任务控制块TCB 的线性地址，以及用户程序的起始扇区号。从图15-15 中可以看出，TCB 线性地址是栈中的第11 个双字（从0 开始算起）。也正是因为如此，TCB 线性地址在栈中的位置是SS：EBP＋44。

同样的道理，用户程序起始逻辑扇区号在栈中的位置是SS：EBP＋48。记好这两个数的位置，一会儿我们就要多次从栈中访问它们。

14.4.3 加载用户程序

当用户程序被读入内存，并处于运行或者等待运行的状态时，就视为一个任务。任务有自己的代码段和数据段（包括栈），这些段必须通过描述符来引用，而这些描述符可以放在GDT 中，也可以放在任务自己私有的LDT 中，但最好是放在LDT 中。GDT 用于存放各个任务公有的描述符，比如公共的数据段和公共例程。

每个任务都允许有自己的LDT，而且可以定义在任何内存位置。所以，我们现在要做三件事：

●分配一块内存，作为LDT 来用，为创建用户程序各个段的描述符做准备；

●将LDT 的大小和起始线性地址登记在任务控制块TCB 中；

●分配内存并加载用户程序，并将它的大小和起始线性地址登记到TCB 中。

第475、476 行，令段寄存器ES 指向4GB 内存段。

第478 行，先从栈中取得TCB 的线性首地址。注意，因为源操作数部分使用的是基址寄存器EBP，故该指令默认使用段寄存器SS 来访问内存（栈）。

接着，第481～484 行，申请分配160 字节的内存空间用于创建LDT，并登记LDT 的初始界限和起始线性地址到TCB 中。LDT 的界限也是16 位的，只允许8192 个描述符。和GDT 一样，界限值是表的总字节数减1，因为我们刚创建LDT，总字节数为0，所以，当前的界限值应当是0xFFFF（0 减去1）。

我们的用户程序很简单，不会划分为太多的段，160 字节的空间可以安装20 个描述符，应当足够了。如图14-12 所示，LDT 的线性起始地址是登记在TCB 内偏移0x0C 处，LDT 的界限是登记在TCB 内偏移0x0A 处。TCB 当初也是动态分配的，需要通过段寄存器ES 指向的4GB段来访问。

第487～500 行，先将用户程序头部读入内核缓冲区中，根据它的大小决定分配多少内存。具体的方法和策略在上一章已讲解过了，唯一需要说明的是，在调用过程sys_routine_seg_sel:read_ hard_disk_0 之前，用户程序的起始逻辑扇区号是从栈中取得的。

第502～504 行，根据用户程序的实际大小申请分配内存空间，并将线性基地址和用户程序的大小登记到TCB 中（参考图14-12）。

一旦知道了用户程序的总大小，接下来，第506～519 行的工作就是加载整个用户程序，这和上一章也是相同的。唯一不同的是，第515 行，从栈中重新取得用户程序的起始逻辑扇区号。

14.4.4　创建局部描述符表

用户程序已被加载到内存中，现在该是在LDT 中创建段描述符的时候了。

第521 行，从TCB 中取得用户程序在内存中的基地址。早在第478 行，我们就已经让ESI寄存器指向了TCB 的基地址。当然，TCB 的基地址位于栈中，也可以从栈中取得。

第524～528 行，因为用户程序头部的起始地址就是整个用户程序的起始地址，故将EDI 寄存器的内容传送到EAX 寄存器，作为过程sys_routine_seg_sel:make_seg_descriptor 的第一个参数，即段的起始地址。接着，从头部中取得用户程序头部段的长度，作为第二个参数传送到EBX 寄存器。因为段界限是段的长度减一，故还要将EBX 寄存器的内容减1。最后，作为第三个参数，在ECX 寄存器中置入段的属性。请

参考段描述符的格式，可以知道，这是一个32 位的可读写数据段，字节粒度，尤其重要的是，其描述符特权级DPL 为3，即最低的特权级。这是可以理解的，谁也不愿意使自己的特权级为3，但这由不得你，谁让你落在操作系统的手上，由它来负责加载呢！

调用过程 sys_routine_seg_sel:make_seg_descriptor 后，会在EDX:EAX 中返回64 位的段描述符。第531、532 行用于调用另一个过程fill_descriptor_in_ldt 把刚才创建的描述符安装到LDT 中。

fill_descriptor_in_ldt 是当前内核代码段的内部（近）过程，位于第421 行，用于在当前任务的LDT 中安装描述符。它需要传入两个参数，一个是要安装的描述符，由EDX:EAX 共同提供；另一个是当前任务控制块的基地址，由EBX 寄存器提供。它用这个地址来访问TCB 以获得LDT 的基地址和当前的大小（界限值），并在安装描述符后更新LDT 的界限值。

第425～428 行，执行例行的现场保护工作，将过程中用到的各个寄存器压栈保护。

第430～433 行，先使段寄存器DS 指向4GB 的内存段；然后，访问TCB，从中取出LDT 的基地址传送到EDI 寄存器。

新描述符的线性地址可以用LDT 的基地址加上LDT 的总字节数得到。第435～440 行，计算用于安装新描述符的线性地址，并把它安装到那里。在这里，ECX 寄存器有两个相关联的用途，一个是在第439 和440 行寻址内存，以安装描述符；另一个是在第436、437 行用于计算LDT 的大小，但只能使用其16 位的CX 部分。想想看，当第一次在LDT 中安装描述符时，LDT 的界限值是0xFFFF，加1 之后，总大小是0x0000，进位部分要丢弃。对CX 寄存器的操作不会影响到ECX 寄存器的高16 位。即使CX 寄存器产生了进位，进位也会丢弃，而决不会跑到ECX 寄存器的高16 位。注意以下指令执行结果的不同：

```
xor ecx,ecx   ;ecx←0
mov cx,0xffff
inc cx        ;ecx=0

xor ecx,ecx   ; ecx←0
mov cx,0xffff
inc ecx       ;ecx=0x00010000
```

和GDT 不同，LDT 的0 号槽位也是可用的。原因在于，其选择子的TI 位是“1”，所以不可能会有一个全零的选择子指向LDT。这就是说，一个指向LDT 的选择子代入段寄存器时，它不可能是因程序员粗心大意而未初始化的。

第442、443 行，将LDT 的总大小（字节数）在原来的基础上增加8字节，再减去1，就是新的界限值。第445 行，将这个新的界限值更新到TCB 中。

第447～450 行，将描述符的界限值除以8，余数丢弃不管，所得的商就是当前新描述符的索引号。

第452～454 行，将CX 寄存器中的索引号逻辑左移3 次，并将TI 位置1，表示指向LDT，这就得到了当前描述符的选择子。

接着回到过程load_relocate_program 中。

过程fill_descriptor_in_ldt 在LDT 中安装描述符后，用CX 寄存器返回一个选择子。第534～536 行，用于将选择子的请求特权级RPL 设置为3，登记到TCB，并回填到用户程序头部。在LDT 中安装的描述符，通常只由用户程序自己使用，即，在请求访问这些段时，请求者是用户程序自己。因此，其选择子的RPL 和用户程序的特权级始终一致。

14.4.5　重定位U-SALT 表

接着回到代码清单14-1 中。

从第539 行开始，一直到第576 行结束，分别是创建用户程序代码段、数据段和栈段描述符，并将它们安装在LDT 中。除了往LDT 中安装描述符，以及其他一些细节上的差别外，这部分代码和上一章相比，大体上是一致的，都很好理解，不需要一一详述。但是，必须要说明的是，在这个过程中所创建的段描述符，其特权级DPL 都是3，而且，这些段描述符的选择子，其请求特权级RPL 也都是3。

从第579 行开始，到第620 行结束，用于重定位用户程序的U-SALT表。和第13 章相比，绝大多数代码都是相同的，具体的工作流程也几乎没有变化。当然，因为涉及特权级，个别的差异还是有的。

U-SALT 位于用户程序头部段。为了访问它，第14 章的做法是先用段寄存器ES 指向用户程序头部段，再访问该段内的U-SALT 表。当然，

前提是用户程序头部段的描述符已经安装并开始生效。

在本章中，用户程序各个段的描述符位于LDT 中，尽管已经安装，但还没有生效（还没有加载局部描述符表寄存器LDTR）。在这种情况下，只能通过4GB 的段来访问U-SALT。所以，第579、580 行用于令段寄存器ES 指向4GB 的内存段。在前面的代码中，是令EDI 寄存器指向用户程序起始加载地址的，这也就是用户程序头部段的起始线性地址。因为U-SALT 的条目数位于头部段内偏移0x24 处。故，程序中用以下指令来取得该条目数（第587 行）：

```
mov ecx,[es:edi+0x24]
```

同样的道理，因为U-SALT 表位于头部段内偏移0x28 处，要想得到U-SALT 表的线性基地址，使EDI 寄存器指向它，程序中使用的是以下指令（第588 行）：

```
add edi,0x28
```

具体的重定位过程在第13 章里已经讲得很清楚了，无非就是找到名字相同的C-SALT 条目，把它的地址部分复制到U-SALT 的对应条目中。在第13 章里，复制的是16 位的代码段选择子和32 位的段内偏移。在本章中，这些地址不再是普通的段选择子和段内偏移，而是调用门选择子和段内偏移。

当初，在创建这些调用门时，选择子的RPL 字段是0。也就是说，这些调用门选择子的请求特权级是0。当它们被复制到U-SALT 中时，应当改为用户程序的特权级（3）。

为此，第605、606 行，因为ESI 寄存器指向当前条目的地址部分，所以4 字节之后的地方是该地址的选择子部分，需要首先传送到AX 寄存器；紧接着，修改它的RPL 字段，使该选择子的请求特权级为3。

14.4.6 创建0、1 和2 特权级的栈

任务在运行时，需要调用内核或者操作系统的例程。这可以认为是从同一个任务的局部地址空间转移到全局地址空间工作。而且，在这个过程中涉及特权级的变化，需要通过调用门。

通过调用门的控制转移通常会改变当前特权级CPL，同时还要切换到与目标代码段特权级相同的栈。为此，必须为每个任务定义额外的栈。对于当前的3 特权级任务来说，应当创建特权级0、1 和2 的栈。而且，应当将它们定义在每个任务自己的LDT 中。

这些额外的栈是动态创建的，而且需要登记在任务状态段（TSS）中，以便处理器固件能够自动访问到它们。但是，现在的问题是还没有创建TSS，有必要先将这些栈信息登记在任务控制块（TCB）中暂时保存。

第622 行，从栈中取得当前任务的TCB 基地址，它是作为过程参数压在当前栈中的。

第625～628 行，申请创建0 特权级栈所需的4KB 内存，并在TCB 中登记该栈的尺寸。登记到TCB 中的尺寸值要求是以4KB 为单位的，所以，还要逻辑右移12 次，相当于除以4096，得到一个4KB 的倍数。

第629、630 行，先申请内存，然后用申请到的内存基地址加上栈的尺寸，得到栈的高端地址，并将此地址登记到TCB 中。一般来说，栈应当使用高端地址作为其线性基地址。

第631、634 行，用给定的段界限和段属性调用公共例程段内的过程make_seg_descriptor 创建描述符。段属性表明这是一个栈段，4KB 粒度。我们创建的是0 特权级栈，故要求描述符的DPL 为0。

第635、636 行，调用内核代码段内的近过程fill_descriptor_in_ldt 将刚创建的描述符安装到LDT 中。该过程要求使用EBX 作为参数提供TCB 的线性基地址，故在调用该过程前先将该地址传送到EBX 寄存器。

第637～639 行，将安装描述符后返回的段选择子登记在TCB 中。相应地，应当将该选择子的请求特权级RPL 设置为0。注意，过程返回的选择子本来就是RPL 为0 的，所以那条指令是作为注释存在的。同时登记的还有0 特权级栈指针的初始值。按老规矩，这个初始值应当为0。

第642～673 行是创建1、2 特权级的栈，并将它们的信息登记在TCB 中，并使用了和上面相同的方法，要注意，为它们分配的特权级别是各不相同的。

14.4.7　安装LDT 描述符到GDT 中

尽管局部描述符表（LDT）和全局描述符表（GDT）都用来存放各种描述符，比如段描述符，但这掩盖不了它们也是内存段的事实。简单地说，它们也是段。但是，因为它们用于系统管理，故称为系统的段或系统段。

全局描述符表（GDT）是唯一的，整个系统中只有一个，所以只需要用GDTR 寄存器存放其线性基地址和段界限即可；但LDT 不同，每个任务一个，所以，为了追踪它们，处理器要求在GDT 中安装每个LDT 的描述符。当要使用这些LDT 时，可以用它们的选择子来访问GDT，将LDT 描述符加载到LDTR 寄存器。在一些人看来，这个理由很牵强，这么做也很别扭。但是，如果不这样，处理器将没有机会来做存储器和特权级的保护工作。

第676～679 行，调用公共例程段的过程make_seg_descriptor 创建LDT 描述符。作为传入的参数，EAX 寄存器的内容是从TCB 中取出的LDT 基地址，EBX 寄存器的内容是从TCB 中取出的LDT 长度，ECX 寄存器的内容是描述符的属性，各属性位与它们在描述符高32 位中相同，无关的位要清零。如图14-16 所示，这是LDT 描述符的格式。

LDT 本身也是一种特殊的段，最大尺寸是64KB。段基地址指示LDT 在内存中的起始地址，段界限指示LDT 的范围；描述符的G 位是粒度位，适用于LDT 描述符，以表示LDT 的界限值是以字节为单位，还是以4KB 为单位。即使是以4KB 为单位，它也不能超过64KB 的大小。

D 位（或者叫B 位）和L 位对LDT 描述符来说没有意义，固定为0。

AVL 和P 位的含义和存储器的段描述符相同。

LDT 描述符中的S 位固定为0，表示系统的段描述符或者门描述符，以相对于存储器的段描述符（S＝1），因为LDT 描述符属于系统的段描述符。



图14-16　LDT 描述符的格式

在描述符为系统的段描述符时，即，在S＝0 的前提下，TYPE 字段为0010（二进制）表明这是一个LDT 描述符。

因此，传送到ECX 寄存器的属性值0x00408200 表示这是一个LDT 描述符，描述符特权级DPL 为0，其他无关的位都已清零。

过程返回后，创建的描述符在EDX:EAX 中。第680 、681 行，立即调用过程set_up_gdt_descriptor 安装此描述符到全局描述符表GDT 中。然后，将返回的描述符选择子写入任务控制块TCB 中的相应位置。

14.4.8　任务状态段TSS 的格式

到目前为止，任务的所有内存段都已创建完毕，除了任务状态段（TSS）。现在就来创建TSS。在此之前，先来全面了解一下TSS 的各个组成部分。

如图14-2 所示，TSS 内偏移0 处是前一个任务的TSS 描述符选择子。和LDT 一样，必须在全局描述符表（GDT）中创建每个TSS 的描述符。当系统中有多个任务同时存在时，可以从一个任务切换到另一个任务执行，此时称任务是嵌套的。被嵌套的任务用这个指针指向前一个任务，即嵌套它的那个任务，当控制返回前一个任务时，处理器需要这个指针来识别前一个任务。创建TSS 时，可以为0。

SS0、SS1 和SS2 分别是0、1 和2 特权级的栈段选择子，ESP0、ESP1 和ESP2 分别是0、1和2 特权级栈的栈顶指针。这些内容应当由任务的创建者填写，且属于填写后一般不变的静态部分，当通过门进行特权级之间的控制转移时，处理器用这些信息来切换栈。

CR3 和分页有关，有关分页的知识将在第16 章讲述。此处一般由任务的创建者填写，如果没有使用分页，可以为0。

偏移为32～92 的区域是处理器各个寄存器的快照部分，用于在进行任务切换时，保存处理器的状态以便将来恢复现场。在一个多任务环境中，每次创建一个任务时，操作系统或者内核至少要填写EIP、EFLAGS、ESP、CS、SS、DS、ES、FS 和GS，当该任务第一次获得执行时，处理器从这里加载初始执行环境，并从CS:EIP 处开始执行任务的第一条指令。在此之后的任务运行期间，该区域的内容由处理器固件

进行更改。在本章中，只有一个任务，而且自进入保护模式时就开始运行了，只不过一开始是在0 特权级的全局空间执行。所以，这部分内容不需要填写。

LDT 段选择子是当前任务的LDT 描述符选择子。由内核或者操作系统填写，以指向当前任务的LDT。该信息由处理器在任务切换时使用，在任务运行期间保持不变。

T 位用于软件调试。在多任务的环境中，如果T 位是"1"，每次切换到该任务时，将引发一个调试异常中断。这是有益的，调试程序可以接管该中断以显示任务的状态，并执行一些调试操作。现在只需要将这一位清零即可。

I/O 映射基地址用于决定当前任务是否可以访问特定的硬件端口，对它的解释说来话长。

是这样的，我们知道，特权指令是只有0 特权级的程序才可以执行的指令，执行这些指令会影响整个机器的状态。

现有的特权指令也许是处理器的设计者精心挑选的，因为即使较低特权级的程序不使用它们，这些程序也能运行得很好，简直是非常好。不过，另外一些候选的指令就没那么幸运了，尽管它们也适合作为特权指令，但其他特权级的程序同样需要它们。

一个典型的例子是硬件端口的输入输出指令in 和out，它们应该对特权级别为1 的程序开放，因为设备驱动程序就工作在这个特权级别。不过，这样做依然是不合理的，因为即使是特权级为3的程序，在需要快速反应的场合，也需要直接访问某些硬件端口。所以，如果需要，它们也可以向2、3 特权级的程序开放。

处理器可以访问65536 个硬件端口。如果只对应用程序开放那些它们需要的端口，而禁止它们访问另一些敏感的端口，操作系统肯定会对此持欢迎态度，因为这有利于设备的统一管理，同时也很安全。

每个任务都有EFLAGS 寄存器的副本，其内容在任务创建的时候由内核或者操作系统初始化，在多任务系统中，每次当任务恢复运行时，就由处理器固件自动从TSS 中恢复。

EFLAGS 寄存器的IOPL 位决定了当前任务的I/O 特权级别。如果当前特权级CPL 高于，或者和任务的I/O 特权级IOPL 相同时，即，在数值上，

```
CPL≤IOPL
```

时，所有I/O 操作都是允许的，针对任何硬件端口的访问都可以通过。

相反，如果当前特权级CPL 低于任务的I/O 特权级IOPL，也并不意味着所有的硬件端口都对当前任务关上了大门。事实上，处理器的意思是总体上不允许，但个别端口除外。至于个别端口是哪些端口，要找到当前任务的TSS，并检索I/O 许可位串。

如图14-17 所示，I/O 许可位串（I/O Permission Bit String）是一个比特序列，或者说是一个比特串，最多允许65536 比特，即8KB。从第1 比特开始，各比特用它在串中的位置代表一个端口号。因此，第1 个比特代表0 号端口，第2 个比特代表1 号端口，第3 个比特代表2 号端口，……，第65536 比特代表第65535 号端口。

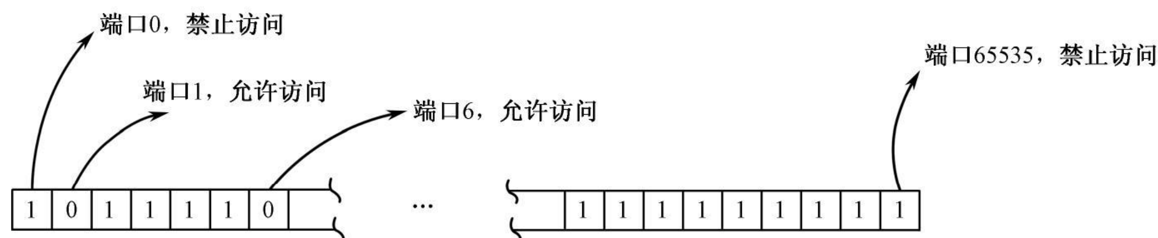


图14-17　最大长度的I/O 许可位串示意图

每个比特的取值决定了相应的端口是否允许访问。为1 时，禁止访问；为0 时，允许访问。

处理器检查I/O 许可位的方法是先计算它在I/O 许可位映射区的字节编号，并读取该字节，然后进行测试。比如，当执行指令

```
out 0x09,al
```

时，处理器通过计算就可以知道，该端口对应着I/O 许可位映射区第2 个字节的第2 个比特（位1）。于是，它读取该字节，并测试那一位。

同其他和任务相关的信息一样，I/O 许可位串位于任务的TSS 中。如图14-18 所示，任务状态段TSS 的最小长度是104 字节，保存着最基本的任务信息，但这并不是它的最大长度。

事实上，整个TSS 还可以包括一个I/O 许可位串，它所占用的区域称为I/O 许可位映射区。如图14-18 所示，在TSS 内偏移为102 的那个字单元，保存着I/O 许可位串（I/O 许可位映射区）的起始位置，从TSS 的起

始处（0）算起。因此，如果该字单元的内容大于或者等于TSS 的段界限（在TSS 描述符中），则表明没有I/O 许可位串。在这种情况下，如果当前特权级CPL 低于当前的I/O 特权级IOPL，执行任何硬件I/O 指令都会引发处理器异常中断。说明一下，和LDT 一样，必须在GDT 中创建TSS 的描述符，TSS 描述符中包括了TSS 的基地址和界限，该界限值包括I/O 许可位映射区在内。

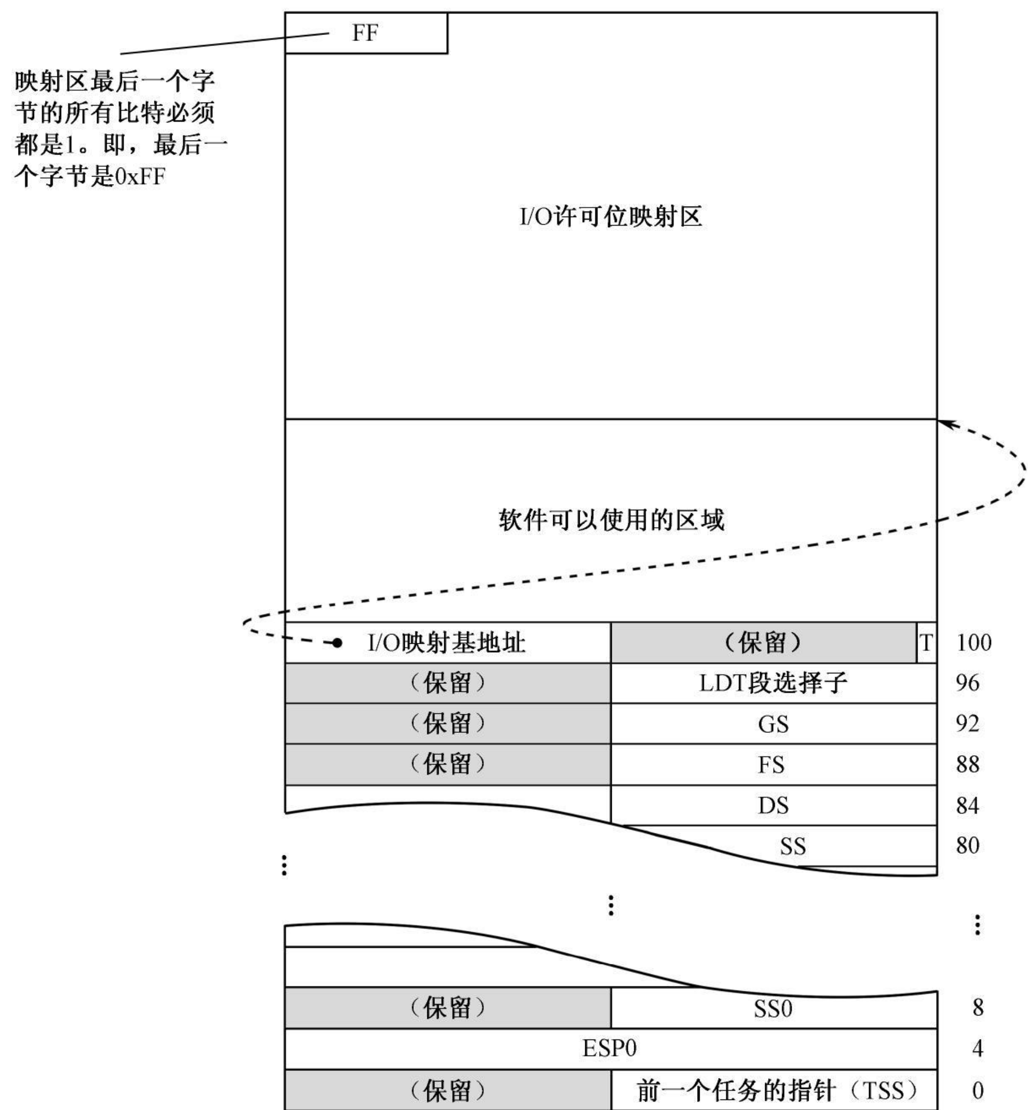


图14-18　TSS 中的I/O 许可位映射区

非常重要的一点是，I/O 端口是按字节编址的。这句话的意思是，每个端口仅被设计用来读写一个字节的数据，当以字或者双字访问时，实际上是访问连续的2 个或者4 个端口。比如，当从端口n 读取一个字时，相当于同时从端口n 和端口n＋1 各读取一个字节。即，

```
in ax,0x3f8
```

相当于同时执行

```
in al,0x3f8
in ah,0x3f9        ;仅为示例，x86 处理器不允许使用 AH 寄存器
```

由于这个原因，当处理器执行一个字或者双字I/O 指令时，会检查许可位串中的2 个，或者4个连续位，而且要求它们必须都是“0”，否则引发异常中断。麻烦在于，这些连续的位可能是跨字节的。即，一些位于前一字节，另一些位于后一字节。为此，处理器每次都要从I/O 许可位映射区读两个连续的字节。

这种操作方式直接导致了另一个问题。即，如果要检查的比特在最后一字节中，那么，这个两字节的读操作将会越界。为防止这种情况，处理器要求I/O 许可位映射区的最后必须附加一个额外的字节，并要求它的所有比特都是“1”，即0xFF。当然，它必须位于TSS 的界限之内。

处理器不要求为每一个I/O 端口都提供位映射。对于那些没有在该区域内映射的位，处理器假定它对应的比特是“1”。例如，要是I/O 许可位映射区的长度是11 字节，那么，除去最后一个所有比特都是“1”的字节，前10 字节映射了80 个端口，分别是端口0 到端口79，访问更高地址的端口将引发异常中断。

显然，EFLAGS 寄存器中的IOPL 位对于控制任务的I/O 特权来说是很重要的。通常，IOPL位由内核或者操作系统根据任务的实际需要进行初始化。尽管不存在对EFLAGS 寄存器整体写入或者读出的指令，但存在将标志寄存器入栈和出栈的指令：

```
pushf/pushfd
popf/popfd
```

pushf 并不是一条新指令。事实上，早在8086 处理器的时代就已经有了，用于将16 位的标志寄存器FLAGS 压栈，机器指令码为9C。在8086 处理器上执行时，SP 寄存器的内容减去2，然后将FLAGS 的内容保存到栈段，操作数的大小是1 个字。同样地，popf 指令把当前栈中的栈顶内容弹出到FLAGS 寄存器。

到了32 位处理器时代，pushf 指令既可以工作在16 位模式下，也可以工作在32 位模式下。在16 位模式下，pushf 压入的是EFLAGS 的低16 位。如果要压入整个32 位的EFLAGS，需要指令前缀66，即

```
66 9C
```

在32 位模式下，pushf 压入的是整个32 位的EFLAGS，即使有指令前缀，也不会只压入低16位，多总比少好，只压入低16 位没有太大意义，徒增处理器的负担。

为了区分EFLAGS 寄存器在16 位模式下的两种压栈方式，编译器引入了符号pushfd。本质上，它们对应着同一条指令，当你使用pushf 时，编译器就知道，应当编译成无前缀的机器码9C；当使用pushfd 时，编译器会编译成66 9C。下面的例子很好地展示了它们之间的区别：

```
[bits 16]
pushf                ;编译后是 9C，16 位操作
pushfd               ;编译后是 66  9C，32 位操作

[bits 32]
pushf                ;编译后是 9C，32 位操作
pushfd               ;编译后同样是 9C，32 位操作
```

可见，在32 位模式下，pushf 和pushfd 是相同的。上面的讨论同样适用于popf 和popfd 指令。

通过将EFLAGS 寄存器的内容压入栈，局部修改后，再弹出到EFLAGS，可以间接地改变它的各种标志位。对多数标志位的修改不会威胁到整个系统的安全，比如，你修改了ZF 标志，这有什么用呢？唯一的后果可能是搬石头砸自己程序的脚。

但是，如果修改了IOPL 位和IF 位，就不同了。能够修改这两个标志的指令是

```
popf  iret  cli  sti
```

注意，没有包括pushf 指令，原因来自一个阴险的想法：你可以执行pushf 指令，但我不允许你执行popf 和iret 指令，你就生气吧！另外，中断是由操作系统或者内核统一管理的，cli 和sti指令不能由低特权级的程

序随便执行。遗憾的是，这些指令并不是特权指令，原因很简单，其他特权级的程序也离不开它们。

最好的办法是用IOPL 本身来控制它们。如果当前特权级CPL 高于，或者和当前I/O 特权级IOPL 相同，即，在数值上

```
CPL≤IOPL
```

则允许执行以上4 条指令，也允许访问所有的硬件端口。否则，如果当前特权级CPL 低于当前的I/O 特权级IOPL，则执行popf 和iret 指令时，会引发处理器异常中断；执行cli 和sti 时，不会引发异常中断，但不改变标志寄存器的IF 位。同时，是否能访问特定的I/O 端口，要参考TSS 中的I/O 许可位映射串。

14.4.9 创建任务状态段TSS

回到代码清单14-1，我们来创建任务状态段TSS。

第684～688 行，申请104 字节的内存用于创建TSS。很显然，我们是要创建一个标准大小的TSS。照例，要把TSS 的基地址和界限登记到任务控制块（TCB）中，将来创建TSS 描述符时用得着。TSS 的界限值是16 位的，是它的大小（总字节数）减1，这就是第686 行的目的。

注意，界限值必须至少是103，任何小于该值的TSS，在执行任务切换时，都会引发处理器异常中断。

第691 行，将指向前一个任务的指针（任务链接域）填写为0，表明这是唯一的任务。

第693～709 行，登记0、1 和2 特权级栈的段选择子，以及它们的初始栈指针。所有的栈信息都在TCB 中，先从TCB 中取出，然后填写到TSS 中的相应位置。

第711、712 行，登记当前任务的LDT 描述符选择子。在任务切换时，处理器需要用这里的信息找到当前任务的LDT。LDT 对任务来说并不是必需的，如果高兴，也可以把属于某个任务的段定义在GDT 中。如果没有LDT，这里应该填写0。

第714、715 行，填写I/O 许可位映射区的地址。在这里，填写的是TSS 段界限（103），这意味着不存在该区域。

14.4.10　安装TSS 描述符到GDT 中

和局部描述符表（LDT）一样，也必须在GDT 中安装TSS 的描述符。这样做，一方面是为了对TSS 进行段和特权级的检查；另一方面，也是执行任务切换的需要。当call far 和jmp far 指令的操作数是TSS 描述符选择子时，处理器执行任务切换操作。

如图14-19 所示，这是TSS 描述符的格式，和LDT 描述符差不多，除了TYPE 位。

TSS 描述符中的B 位是“忙”位（Busy）。在任务刚刚创建的时候，它应该为二进制的1001，即，B 位是0，表明任务不忙。当任务开始执行时，或者处于挂起状态（临时被中断执行）时，由处理器固件把B 位置1。

任务是不可重入的。就是说，在多任务环境中，如果一个任务是当前任务，它可以切换到其他任务，但不能从自己切换到自己。在TSS 描述符中设置B 位，并由处理器固件进行管理，可以防止这种情况的发生。



图14-19　TSS 描述符的格式

第720～725 行，先调用公共例程段内的过程make_seg_descriptor 创建TSS 描述符，它需要传入三个参数。先从TCB 中取出TSS 的基地址，传送到EAX 寄存器；EBX 寄存器的内容是TSS 的界限；ECX 寄存器的内容是描述符属性值，0x00408900 表明这是一个DPL 为0 的TSS 描述符，字节粒度。接着，调用公共例程段内的另一个过程set_up_gdt_descriptor 安装此描述符到GDT中，并将返回的描述符选择子登记在TCB 中。TSS 描述符选择子的RPL 字段为0。

14.4.11　带参数的过程返回指令

至此，任务创建完毕，可以从过程load_relocate_program 返回了。

在过程返回之前，即，在执行ret 指令之前，需要恢复现场，也就是按相反的顺序将刚进入过程时压入栈的内容出栈。这是第727～730 行的工作。

如图14-20 所示，当执行ret 指令时，栈恢复到刚进入过程时的状态，即，只有返回地址和调用者传递给过程的参数。因为当初是采用32 位相对近调用进入过程load_relocate_program 的，故仅将EIP 压栈，没有压入段寄存器CS 的内容。

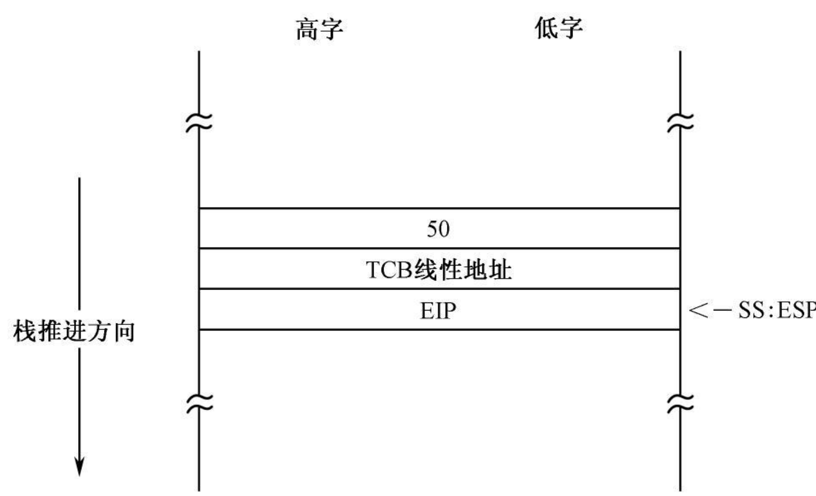


图14-20　执行ret 指令时的栈状态

再来看，一旦ret 指令执行完毕，控制将返回到调用者，且栈中只剩下两个参数。按道理，这两个参数是由调用者压入的，应该再由调用者弹出即可：

```
push dword 50                          ;用户程序位于逻辑 50 扇区
push ecx                               ;压入任务控制块起始线性地址
call load_relocate_program             ;调用过程
```

```
add esp,8                              ;过程返回后，调整栈指针使之越过参数
```

不过，最好的解决办法是在过程返回时，顺便弹出参数。这样做是可行的，过程的编写者最清楚栈中有几个参数。如果希望过程在返回时

弹出参数，使ESP 寄存器指向调用过程前的栈位置（使栈平衡），可以使用带操作数的过程返回指令：

```
ret imm16
retf imm16
```

这两条指令都允许16 位的立即数作为操作数，不同之处仅仅在于，前者是近返回，后者是远返回。立即数是16 位的，而且一般总是偶数，原因是栈操作总是以字或者双字进行，它指示在将控制返回到调用者之前，应当从栈中弹出多少字节的数据。

因此，第732 行，当该指令执行时，除了将控制返回到过程的调用者之外，还要调整栈的指针，即

```
ESP←ESP＋8
```

之所以指令的操作数是8，是因为要弹出2 个双字。

这条指令给高级语言带来的好处是增加了它们的复杂性。比如这样一个C 语言函数：

```
void func(int i,char *c) {
/* 这里是函数体 */
}
```

因为一般是通过栈传递参数，所以，哪个参数先入栈，哪个后入栈，栈平衡的事情由调用者来做，还是由过程来做，就需要一个标准，即所谓的调用转换规则。特别是在开发一些大软件时，需要用不同的高级语言来开发各个独立的、但能够协同工作的模块，尤其需要注意这个问题。

一个典型的调用转换标准是stdcall，它规定，参数从右往左进栈，且由过程在返回前出栈。

14.5 用户程序的执行

14.5.1 通过调用门转移控制的完整过程

现在我们转到代码清单14-1 的第845、846 行，在调用过程load_relocate_program 创建任务之后，显示一条成功的消息。

接下来的工作是将控制转到用户程序那里。我们创建的是一个3 特权级的任务，所以这是一个从0 特权级到3 特权级的控制转移。或者，换一种更体面的说法，是从任务自己的0 特权级全局空间转移到3 特权级局部空间执行。通常情况下，这既不允许，也不太可能。

办法总还是有的，只不过稍微有一点曲折，那就是假装从调用门返回。先来看看完整的调用门控制转移和返回过程是怎样的。

首先，通过调用门实施控制转移，可以使用jmp far 和call far 指令。指令执行时，描述符选择子必须指向调用门，32 位偏移量被忽略。但，无论采用哪种控制转移指令，都会使用表14-1 的特权检查规则。注意，表中的比较关系都是数值上的。

表14-1　调用门的特权级检查规则

指　　令	特权检查规则
CALL FAR	CPL≤调用门描述符的 DPL，RPL≤调用门描述符的 DPL 对于依从和非依从的代码段：CPL≥目标代码段描述符的 DPL
JMP FAR	CPL≤调用门描述符的 DPL，RPL≤调用门描述符的 DPL 若目标代码段是依从的：CPL≥目标代码段描述符的 DPL 若目标代码段是非依从的：CPL＝目标代码段描述符的 DPL

从表15-1 中可以看出，当使用jmp far 指令通过调用门转移控制时，要求当前特权级和目标代码段的特权级相同。原因是用jmp far 指令通过调用门转移控制时，不改变当前特权级CPL。

相反，使用call far 指令可以通过调用门将控制转移到较高特权级别的代码段。之所以说“可以”，是因为，如果目标代码段是依从的，则和jmp far 指令一样，不改变当前特权级别；否则，如果目标代码段是非依从的，则在目标代码段的特权级别上执行。

其次，当使用call far 指令通过调用门转移控制时，如果改变了当前的特权级别，则必须切换栈。即，从当前任务的固有栈切换到与目标代码段特权级相同的栈上。栈的切换是由处理器固件自动进行的。

当前栈是由段寄存器SS 和栈指针寄存器ESP 的当前内容指示的；要切换到的新栈位于当前任务的TSS 中，处理器知道如何找到它。在栈切换前，处理器要检查新栈是否有足够的空间完成本次控制转移。栈切换过程如下：

① 使用目标代码段的DPL（也就是新的CPL）到当前任务的TSS 中选择一个栈，包括栈段选择子和栈指针。

② 从TSS 中读取所选择的段选择子和栈指针，并用该选择子读取栈段描述符。在此期间，任何违反段界限检查的行为都将引发处理器异常中断（无效TSS）。

③ 检查栈段描述符的特权级和类型，并可能引发处理器异常中断（无效TSS）。

④ 临时保存当前栈段寄存器SS 和栈指针ESP 的内容。

⑤ 把新的栈段选择子和栈指针代入SS 和ESP 寄存器，切换到新栈。

⑥ 将刚才临时保存的SS 和ESP 的内容压入当前栈，如图14-21 所示。

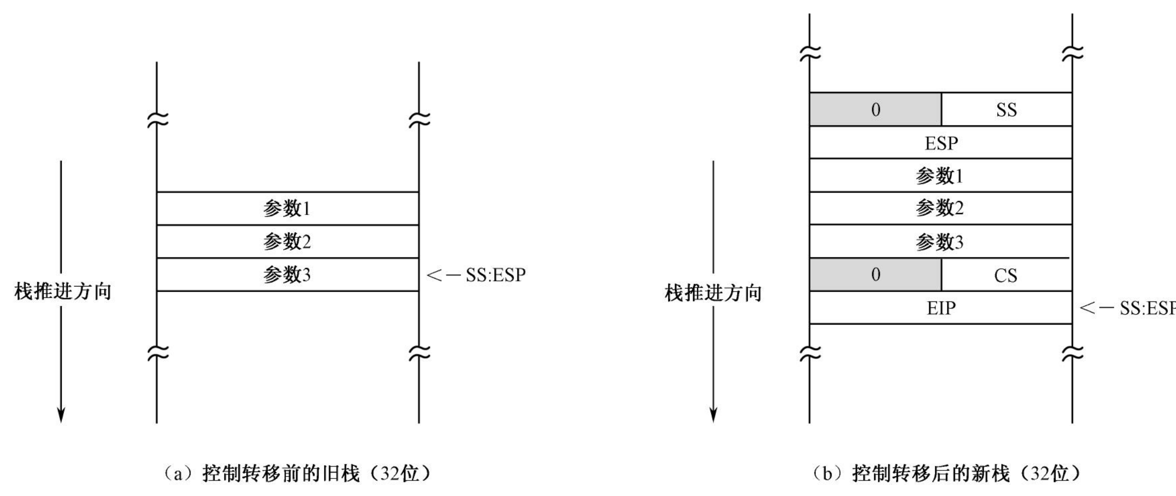


图14-21　特权级间控制转移时的栈切换

⑦ 依据调用门描述符“参数个数”字段的指示，从旧栈中将所有参数都复制到新栈中。如果参数个数为0，不复制参数，如图14-21 所示。

⑧ 将当前段寄存器CS 和指令指针寄存器EIP 的内容压入新栈，如图14-21 所示。通过调用门实施的控制转移一定是远转移，所以要压入CS和EIP。

⑨ 从调用门描述符中依次将目标代码段选择子和段内偏移传送到CS和EIP 寄存器，开始执行被调用过程。

相反，如果没有改变特权级别，则不切换栈，继续使用调用者的当前栈，只在原来的基础上压入当前段寄存器CS 和指令指针寄存器EIP 的内容，如图14-22 所示。

再次，如果通过调用门的控制转移是使用jmp far 指令发起的，结果就是肉包子打狗，有去无回。而且，没有特权级的变化，也不需要切换栈。相反，如果通过调用门的控制转移是使用call far指令发起的，那么，可以使用远返回指令retf 把控制返回到调用者。

从同一特权级返回时，处理器将从栈中弹出调用者的代码段选择子和指令指针。尽管它们通常是有效的，但是，为了安全起见，处理器依然会进行特权级检查。

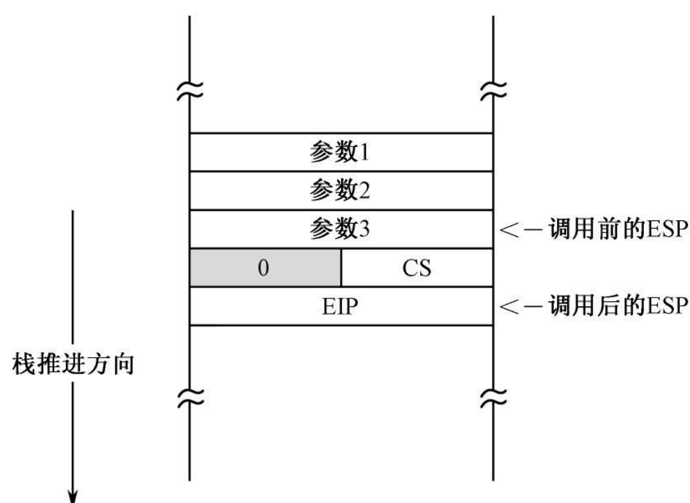


图14-22　相同特权级控制转移前后的栈变化

要求特权级变化的远返回，只能返回到较低的特权级别上。控制返回的全部过程如下：

① 检查栈中保存的CS 寄存器的内容，根据其RPL 字段决定返回时是否需要改变特权级别。

② 从当前栈中读取CS 和EIP 寄存器的内容，并针对代码段描述符和代码段选择子的RPL 字段实施特权级检查。

③ 如果远返回指令是带参数的，则将参数和ESP 寄存器的当前值相加，以跳过栈中的参数部分。最后的结果是ESP 寄存器指向调用者SS 和ESP 的压栈值。注意，retf 指令的字节计数值必须等于调用门中的参数个数乘以参数长度。

④ 如果返回时需要改变特权级，从栈中将SS 和ESP 的压栈值代入段寄存器SS 和指令指针寄存器ESP，切换到调用者的栈。在此期间，一旦检测到有任何界限违例的情况都将引发处理器异常中断。

⑤ 如果远返回指令是带参数的，则将参数和ESP 寄存器的当前值相加，以跳过调用者栈中的参数部分。最后的结果是调用者的栈恢复到平衡位置。

⑥ 如果返回时需要改变特权级，检查DS、ES、FS 和GS 寄存器的内容，根据它们找到相应的段描述符。要是有任何一个段描述符的DPL 高于调用者的特权级（返回后的新CPL），即，在数值上，那么，处理器将把数值0 传送到该段寄存器。

```
段描述符的 DPL＜返回后的新 CPL
```

那么，这是为什么呢？

特权级检查不是在实际访问内存时进行的，而是在将选择子代入段寄存器时进行的。下面这两条指令可以非常清楚地说明这一点：

```
mov ds,ax                  ;进行特权级检查
mov edx,[0x2000]           ;不进行特权级检查
```

要想访问内存中的数据，必须先指定一个段。即，将选择子代入某个段寄存器。正是因为如此，处理器只在将选择子代入段寄存器时进行一次特权级检查，而在此之后的普通内存访问时，不进行特权级检查。处理器的意思是，只要你能进入大门，就证明你的确是这里的主人，随后你干什么它都不会干涉。

现在做一个假设，假设一个3 特权级的应用程序通过调用门请求0 特权级的操作系统服务。在进入操作系统例程后，当前特权级CPL 变成0。在该例程内，操作系统可能会访问自己的0 特权级数据段以进行某些内部操作。当然，它也必须先执行将选择子代入段寄存器的操作：

```
mov ds,ax                    ;操作系统自己的选择子
```

按道理，安全的做法是先将旧的DS 值压栈，用完后再出栈。像这样：

```
push ds
mov ds,ax
……
pop ds
retf
```

但是，如果操作系统例程没有这么做，一定有它的道理，而处理器也无权干涉。唯一可以预料的是，当控制返回到应用程序时，段寄存器DS 依然指向操作系统数据段。因此，应用程序就可以直接在3 特权级下访问操作系统的数据段：

```
mov edx,[0x000c]
```

这是因为，特权级检查只在引用一个段的时候进行。即，只在将选择子传送到段寄存器的时候进行。只要通过了这一关，后面那些使用这个段寄存器的内存访问就都是合法的。

为了解决这个问题，在执行retf 指令时，要检查数据段寄存器，根据它们找到相应的段描述符。要是有任何一个段描述符的DPL 高于调用者的特权级（返回后的新CPL），那么，处理器将把数值0 传送到该段寄存器。使用这样的段寄存器访问内存，会引发处理器异常中断。

特别需要注意的是，任务状态段（TSS）中的SS0、ESP0、SS1、ESP1、SS2、ESP2 域是静态的，除非软件进行修改，否则处理器从来不会改变它们。举个例子，当处理器通过调用门进入0特权级的代码段时，会切换到0 特权级栈。返回时，并不把0 特权级栈指针的内容更新到TSS 中的ESP0 域。下次再次通过调用门进入0 特权级代码段时，使用的依然是ESP0 的静态值，从来不会改变。这就是说，如果你希望通过0

特权级栈返回数据，就必须自己来做这件事，比如，在返回到低特权级别的代码段之前，手工改写TSS 中的ESP0 域。

14.5.2　进入3 特权级的用户程序的执行

接着回到代码清单14-1 中。

任务寄存器TR 总是指向当前任务的任务状态段（TSS），而LDTR 寄存器也总是指向当前任务的LDT。TSS 是任务的主要标志，因此要使TR 寄存器指向任务；而使用LDTR 的原因是可以在任务执行期间加速段的访问。

在多任务环境中，随着任务的切换，每当一个任务开始运行时（成为前台活动任务），TR 和LDT 寄存器的内容都会更新，以指向新的当前任务。

现在的问题是，我们只有一个任务，而且是个3 特权级的任务，不能用任务切换的方法使它开始运行。这个问题可以表述为：如何从任务的0 特权级全局空间转移到它自己的3 特权级空间正常执行？

答案是先确立身份，即，使TR 和LDTR 寄存器指向这个任务，然后假装从调用门返回。和当前任务有关的信息都在它的任务控制块（TCB）中。因此，第832、833 行，先令段寄存器DS指向4GB 的内存段。

第851、852 行，加载任务寄存器TR 和局部描述符表寄存器（LDTR）。

如图14-23 所示，TR 和LDTR 寄存器都包括16 位的选择器部分，以及的描述符高速缓存器部分。选择器部分的内容是TR 和LDT 描述符的选择子；描述符高速缓存器部分的内容则指向当前任务的TSS 和LDT，以加速这两个段（表）的访问。

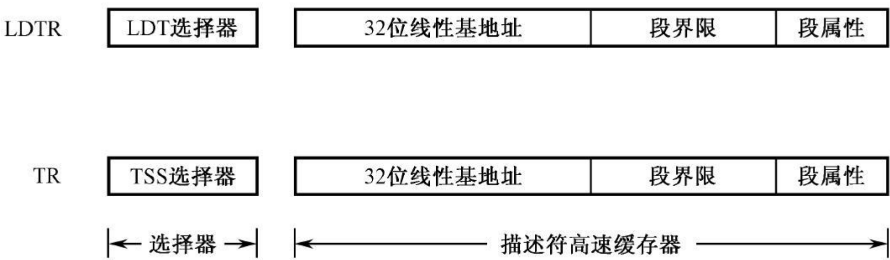


图14-23　LDTR 和TR 寄存器

加载任务寄存器TR 需要使用ltr 指令。这条指令的格式为

```
ltr r/m16
```

这条指令的操作数可以是16 位通用寄存器，也可以是指向一个16 位单元的内存地址。但不管是寄存器还是内存单元，其内容都是16 位的TSS 选择子。

在将TSS 选择子加载到TR 寄存器之后，处理器用该选择子访问GDT 中对应的TSS 描述符，将段界限和段基地址加载到任务寄存器TR 的描述符高速缓存器部分。同时，处理器将该TSS 描述符中的B 位置“1”，也就是标志为“忙”，但并不执行任务切换。

该指令不影响EFLAGS 寄存器的任何标志位，但属于只能在0 特权级下执行的特权指令。

加载局部描述符表寄存器（LDTR）使用的是lldt 指令，其格式和ltr 是一样的：

```
lldt r/m16
```

其操作数也和ltr 指令一样，但是，指向的是16 位LDT 选择子。ltr 和lldt 指令执行时，处理器首先要检查描述符的有效性，包括审查它是不是TSS 或者LDT 描述符。在将LDT 选择子加载到LDTR 寄存器之后，处理器用该选择子访问GDT 中对应的LDT 描述符，将段界限和段基地址加载到LDTR 的描述符高速缓存器部分。CS、SS、DS、ES、FS 和GS 寄存器的当前内容不受该指令的影响，包括TSS 中的LDT 选择子字段。

如果执行这条指令时，代入LTR 选择器的选择子，其高14 位是全零，LDTR 寄存器的内容被标记为无效，而该指令的执行也将不声不响地结束（即不会引发异常中断）。当然，后续那些引用LDT 的指令都将引发处理器异常中断（对描述符进行校验的指令除外），例如，将一个指向LDT的段选择子代入段寄存器。

最后，如图14-24 所示，这是一个任务的全景图，给出了与一个任务相关的各个组成部分。

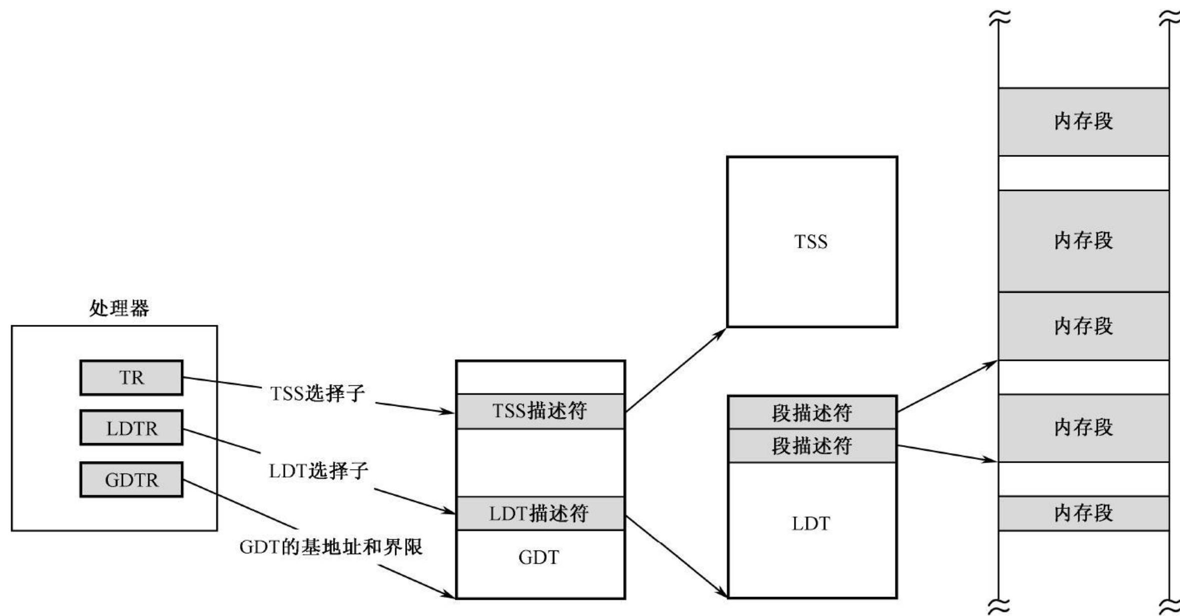


图14-24　与任务相关的各部分逻辑关系示意图

注意了，现在，局部描述符表（LDT）已经生效，可以通过它访问用户程序的私有内存段了。

第854、855 行，访问任务的TCB，从中取出用户程序头部段选择子，并传送到段寄存器DS。该选择子RPL 字段的值为3，即，请求特权级为3；TI 位是“1”，指向任务自己的LDT。这两条指令执行后，段寄存器DS 就指向用户程序头部段。

第858～862 行，从用户程序头部内取出栈段选择子和栈指针，以及代码段选择子和入口点，并将它们顺序压入当前的0 特权级栈中。这部分内容要结合第13 章的用户程序头部来分析（代码清单13-3）。

第864 行，执行一个远返回指令retf，假装从调用门返回。于是控制转移到用户程序的3 特权级代码开始执行。注意，这里所用的0 特权级栈并非是来自于TSS。不过，处理器不会在意这个。下次，从3 特权级的段再次来到0 特权级执行时，就会用到TSS 中的0 特权级栈了。

现在回到上一章，看代码清单13-3。

用户程序现在是工作在它的局部空间里。它可以通过调用门请求系统服务来显示字符串，或者读取硬盘数据，这都没有问题。这些指令可以再次加深我们对调用门的理解，请读者自行分析。

唯一的问题是，当它最后用jmp far 指令将控制权返回到内核时，可能行不通了。这条指令是

```
jmp far [fs:TerminateProgram]      ;将控制权返回到系统
```

这确实是一个调用门。而且，通过jmp far 指令使用调用门也没有任何问题。问题在于，当控制转移到内核时，当前特权级没有变化，还是3，因为使用jmp far 指令通过调用门转移控制是不会改变当前特权级别的。

再回到本章，看代码清单14-1。

返回点是在第866 行。因为当前特权级是3，以这样低的特权级别来执行第867、868 行的指令，一定会引发处理器异常中断:

```
mov eax,core_data_seg_sel
mov ds,eax
```

在这里，当前特权级CPL 为3，选择子core_data_seg_sel 的请求特权级RPL 为0，目标代码段的特权级DPL 为0，因为当前特权级CPL 低于目标代码段的DPL，就算请求特权级RPL 和目标代码段的DPL 相同，也不可能通过特权级检查。

异常和异常中断的处理将在第17 章讲述，我们现在还没有任何接管和处理异常中断的机制，所以，这个异常可能不会明显地被你观察到。

还需要特别提醒的是，进入3 特权级的用户程序局部空间时，任务的I/O 特权级IOPL 是0，任务没有I/O 操作的特权。

最后，将本章的源代码编译，并从第1 个逻辑扇区开始，将编译后的文件写入虚拟硬盘。如果你用的虚拟硬盘文件还是第13 章用过的那个，这就是唯一要做的工作；否则，还要写入第13章的主引导程序、用户程序和数据文件。具体方法参见上一章。最后，启动虚拟机，应该能观察到如图14-25 所示的画面。

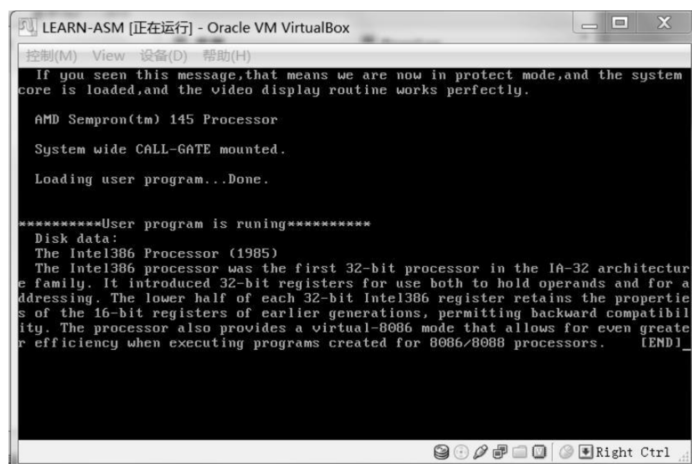


图14-25　本章程序的运行结果

14.5.3　检查调用者的请求特权级RPL

在本章的最后，我们回过头来聊一聊与请求特权级RPL 有关的问题。通过这个话题的深入，你会更进一步了解处理器引入RPL 的原因和意义。

为了访问一个段，首先需要将段选择子代入段寄存器，这也是处理器进行特权级检查的大好机会：

```
mov fs,cx
```

在绝大多数情况下，请求访问一个段的程序也是段选择子的提供者。就是说，当前特权级和请求特权级是相同的，即，RPL＝CPL。

一般来说，用户程序的特权级别很低，而且不能执行I/O 操作。假设操作系统提供了一个例程，可以从用户程序那里接受三个参数：逻辑扇区号、数据段选择子和段内偏移量，然后读硬盘，并把数据传送到用户程序的缓冲区内。为了使用户程序可以调用此例程，操作系统把它定义成调用门。

一般来说，用户程序会提供一个RPL 为3 的段选择子给操作系统例程。通过调用门实施控制转移后，当前特权级CPL 变成0，实际的请求者是用户程序，选择子的请求特权级RPL 为3，要访问的段属于用户程序，其描述符的DPL 为3，在数值上符合CPL≤DPL，并且RPL≤DPL 的条件，可以正常执行。

人类的可恶之处无孔不入，总爱钻空子。想象一下，用户程序的编写者通过钻研，知道了内核数据段的选择子，而且希望用这个选择子访问内核数据段。当然，他不可能在用户程序里访问内核数据段，因为那个数据段的DPL 为0，而用户程序工作时的当前特权级为3，处理器会很机警地把来访者拒之门外。

但是，他可以借助于刚才那个调用门。特别是，他提供的是一个RPL 为0 的选择子，而且该选择子指向操作系统的段描述符。此时，当前特权级CPL 为0，请求特权级RPL 为0，目标数据段描述符的DPL 为0，同样符合在数值上符合CPL≤DPL，并且RPL≤DPL 的条件，并且允许向内核数据段写入扇区数据，他得逞了！

我知道，有人会说，通过调用门进入内核例程时，用户程序的代码段选择子就作为返回地址压在栈中，代码段选择子的低2 位就是用户程序的特权级。因此，可以改造处理器固件，使它能够访问栈，用这个特权级来进行特权级检查。

但是，有这种认识的朋友们忘了，处理器的智商很低，它不可能知道谁是真正的请求者。你当然可以通过分析程序的行为来区分它们，但处理器不能。因此，当指令

```
mov ds,ax
```

或者

```
mov ds,cx
```

执行时，AX 或者CX 寄存器中的选择子可能是内核自己提供的，也可能来自于恶意的用户程序，是不是合法，这两种情况要区别对待，不能一棍子打死。所以，这已经超出了处理器的能力和职权范围。

怎么办？

还记得在本章的前面，在讨论RPL 时，我是怎么说的？我说的是，RPL 只是在原来的基础上多增加了一种检查机制，并把如何能够通过这种检查的自由裁量权交给软件（的编写者）。

引入请求特权级RPL 的原因是处理器在遇到一条将选择子传送到段寄存器的指令时，无法区分真正的请求者是谁。但是，引入RPL 本身并不能完全解决这个问题，这只是处理器和操作系统之间的一种协议，处

理器负责检查请求特权级RPL，判断它是否有权访问，但前提是提供了正确的RPL；内核或者操作系统负责鉴别请求者的身份，并有义务保证RPL 的值和它的请求者身份相符，因为这是处理器无能为力的。

因此，在引入RPL 这件事上，处理器的潜台词是，仅依靠现有的CPL 和DPL，无法解决由请求者不同而带来的安全隐患。那么，好吧，再增加一道门卫，但前提是，操作系统只将通行证发放给正确的人。

为了帮助内核或者操作系统核查请求者的身份，并提供正确的RPL值，处理器提供了arpl 指令。arpl 指令的作用是调整段选择子RPL 字段的值（Adjust RPL Field of Segment Selector），其格式为

```
arpl r/m16,r16
```

该指令比较两个段选择子的RPL 字段，目的操作数可以是包含了16 位段选择子的通用寄存器，或者指向一个16 位单元的内存地址，该字单元里存放的是段选择子；源操作数只能是包含了段选择子的16 位通用寄存器。

该指令执行时，处理器检查目的操作数的RPL 字段，如果它在数值上小于源操作数的RPL 字段，则设置ZF 标志，并增加目的操作数RPL 字段的值，使之和源操作数RPL 字段的值相同。否则，ZF 标志清零，而且除此之外什么也不会发生。

arpl 是典型的操作系统指令，它通常用于调整应用程序传递给操作系统的段选择子，使其RPL 字段的值和应用程序的特权级相匹配。在这种情况下，传递给操作系统的段选择子是作为目的操作数出现的；而应用程序的段选择子是作为源操作数出现的（可以从栈中取得）。arpl 也可以在应用程序中使用。

这样，为了防止恶意的数据访问，操作系统应该从当前栈中取得用户程序的代码段选择子（调用者代码段寄存器CS 的内容）作为源操作数，并把作为参数传递进来的数据段选择子作为目的操作数，来执行arpl指令，把数据段选择子的请求特权级RPL 调整（恢复）到调用者的特权级别上。

一旦调整了请求特权级，那么，当前特权级CPL 为0，请求特权级RPL 为3，数据段描述符特权级DPL 为0，数值上并不符合CPL≤DPL，并且RPL≤DPL 的条件，禁止访问，并引发处理器异常中断。

引入RPL 检查机制和arpl 指令，主要是防止对段的不安全访问，不管是恶意的，还是因为编程疏漏而引起的。不管怎么说，一旦引入了RPL 检查机制，它就会处处起作用，同时也就成了编写程序时不得不考虑和妥善处理的问题。

14.5.4　在Bochs 中调试程序的新方法

随着本书内容的深入，程序会越来越复杂，但一般不会出什么问题，因为我都调试好了。当然，你可能想在此基础上做一些改动，实现其他一些功能。在这种情况下，每一次运行时能得到预期结果的可能性微乎其微。不过不用担心，使用本书前面讲过的调试技术，你一定能够找到问题所在。比如，你可以使用“info gdt”指令察看GDT 中的段描述符和门描述符。

本章中涉及到两个新的系统寄存器LDTR 和TR，要察看它们的内容，可以使用以前讲过的Bochs 调试指令“sreg”；为了察看局部描述符表LDT 的所有内容，可以使用“info ldt”；要察看任务状态段TSS 的内容，可以使用“info tss”。注意，显示LDT 和TSS 的内容时，Bochs 要先从处理器的TR 和LDTR 寄存器获取基地址和界限信息。因此，显示的是当前任务的LDT 和TSS。如图14-26 所示，这里显示了在执行代码清单14-1 的第851 行“ltr [ecx+0x18]”之后，用“info tss”命令显示的TSS 状态。

```
Bochs for Windows - Console
(0) [0x000000000004141f] 0038:000000000000049f (unk. ctxt): mov ds, ax
      ; 8ed8
<bochs:110> n
Next at t=17860191
(0) [0x0000000000041421] 0038:00000000000004a1 (unk. ctxt): ltr word ptr ds:[ecx
+24]  ; 0f005918
<bochs:111> n
Next at t=17860192
(0) [0x0000000000041425] 0038:00000000000004a5 (unk. ctxt): lldt word ptr ds:[ec
x+16] ; 0f005110
<bochs:112> info tss
tr:s=0x68, base=0x00000000001048e8, valid=1
ss:esp(0): 0x0024:0x00000000
ss:esp(1): 0x002d:0x00000000
ss:esp(2): 0x003e:0x00000000
cr3: 0x00000000
eip: 0x00000000
eflags: 0x00000000
cs: 0x0000 ds: 0x0000 ss: 0x0000
es: 0x0000 fs: 0x0000 gs: 0x0000
eax: 0x00000000  ebx: 0x00000000  ecx: 0x00000000  edx: 0x00000000
esi: 0x00000000  edi: 0x00000000  ebp: 0x00000000  esp: 0x00000000
ldt: 0x0060
i/o map: 0x0067
<bochs:113> _
```

图14-26　用info tss 命令显示TSS 段的内容

检测点14.3

如图14-26，为什么该任务的TSS 中，所有段寄存器和通用寄存器的值都是0 而不影响任务的执行？

本章习题

1．修改代码清单14-1 和13-3，使用户程序能够正常返回到内核，并在显示消息后停机。

2．修改代码清单14-1 和13-3，使得通过调用门请求读取硬盘扇区的服务时，通过栈传递参数。而且，传递的参数分别是逻辑扇区号、数据段选择子和段内偏移。要求使用arpl 指令。

第15章　任务切换

从80286 开始的处理器是面向多任务系统而设计的。在一个多任务的环境中，可以同时存在多个任务，每个任务都有各自的局部描述符表（LDT）和任务状态段（TSS）。在局部描述符表中存放着专属于任务局部空间的段的描述符。可以在多个任务之间切换，使它们轮流执行，从一个任务切换到另一个任务时，具体的切换过程是由处理器固件负责进行的。

所谓多任务系统，是指能够同时执行两个以上任务的系统。即使前一个任务没有执行完，下一个任务也可以开始执行。但是，什么时候切换到另一个任务，以及切换到哪一个任务执行，主要是操作系统的责任，处理器只负责具体的切换过程，包括保护前一个任务的现场。

有两种基本的任务切换方式，一种是协同式的，从一个任务切换到另一个任务，需要当前任务主动地请求暂时放弃执行权，或者在通过调用门请求操作系统服务时，由操作系统“趁机”将控制转移到另一个任务。这种方式依赖于每个任务的“自律”性，当一个任务失控时，其他任务可能得不到执行的机会。

另一种是抢占式的，在这种方式下，可以安装一个定时器中断，并在中断服务程序中实施任务切换。硬件中断信号总会定时出现，不管处理器当时在做什么，中断都会适时地发生，而任务切换也就能够顺利进行。在这种情况下，每个任务都能获得平等的执行机会。而且，即使一个任务失控，也不会导致其他任务没有机会执行。

抢占式多任务将在第17 章讲解，本章先介绍多任务任务切换的一般工作原理，掌握任务切换的几种方法，以及它们各自的特点。

15.1 本章代码清单

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：15-1（保护模式微型核心程序），源程序文件：c15_core.asm

本章代码清单：15-2（动态加载的用户程序），源程序文件：c15.asm

15.2　任务切换前的设置

在上一章里，有关特权级间的控制转移落墨较多，容易使读者混淆了它和任务切换之间的区别。如图15-1 所示，所有任务共享一个全局空间，这是内核或者操作系统提供的，包含了系统服务程序和数据；同时，每个任务还有自己的局部空间，每个任务的功能都不一样，所以，局部空间包含的是一个任务区别于其他任务的私有代码和数据。

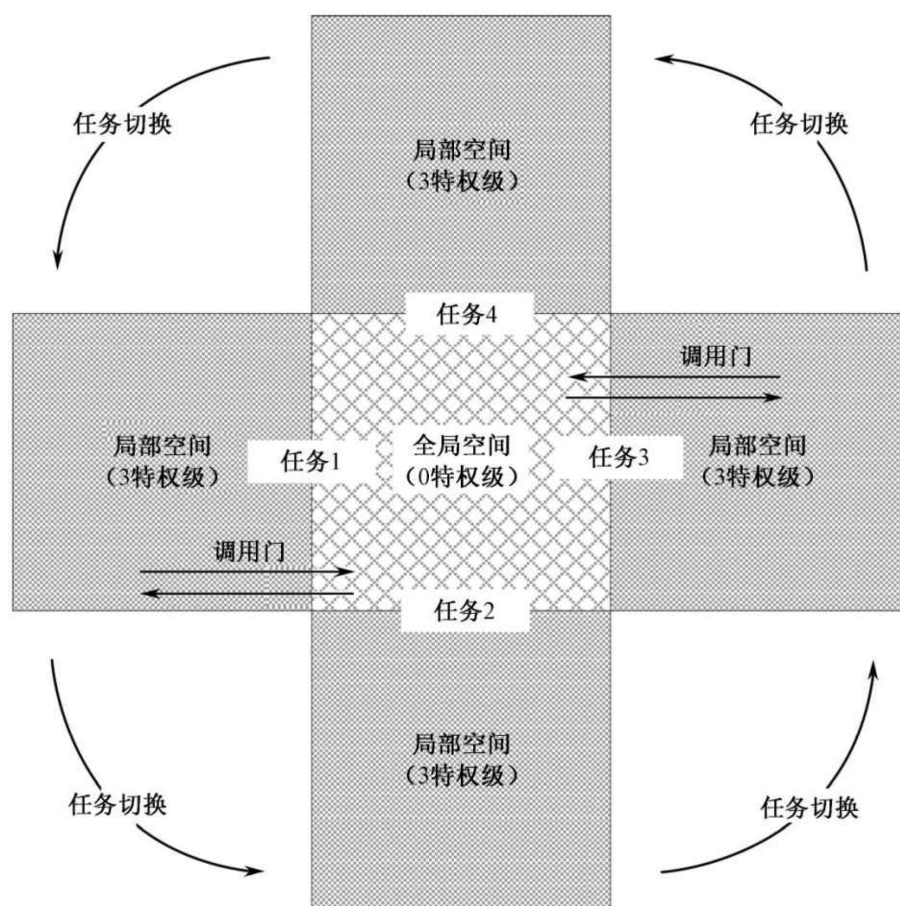


图15-1　任务切换和任务内特权级间的控制转移

在一个任务内，全局空间和局部空间具有不同的特权级别。使用门，可以在任务内将控制从3特权级的局部空间转移到0 特权级的全局空间，以使用内核或者操作系统提供的服务。

任务切换是以任务为单位的，是指离开一个任务，转到另一个任务中去执行。任务转移相对来说要复杂得多，当一个任务正在执行时，处理器的各个部分都和该任务息息相关：段寄存器指向该任务所使用的内存段；通用寄存器保存着该任务的中间结果，等等。离开当前任务，转到另一个任务开始执行时，要保存旧任务的各种状态，并恢复新任务的运行环境。

这就是说，要执行任务切换，系统中必须至少有两个任务，而且已经有一个正在执行中。在上一章中，我们已经创建过一个任务，那个任务的特权级别是3，即最低的特权级别。一开始，处理器是在任务的全局空间执行的，当前特权级别是0，然后，我们通过一个虚假的调用门返回，使处理器回到任务的局部空间执行，当前特权级别降为3。

事实上，这是没有必要的，这样做很别扭。首先，处理器在刚进入保护模式时，是以0 特权级别运行的，而且执行的一般是操作系统代码，也必须是0 特权级别的，这样才能方便地控制整个计算机。其次，任务并不一定非得是3 特权级别的，也可以是0 特权级别的。特别是，操作系统除了为每一个任务提供服务外，也会有一个作为任务而独立存在的部分，而且是0 特权级别的任务，以完成一些管理和控制功能，比如提供一个界面和用户进行交互。

既然是这样，当计算机加电之后，一旦进入保护模式，就直接创建和执行操作系统的0 特权级任务，这既自然，也很方便。然后，可以从该任务切换到其他任务，不管它们是哪个特权级别的。

既然如此，我们在这一章里就要首先创建0 特权级别的操作系统（内核）任务。

本章同样没有主引导程序，还要使用第13 章的主引导程序，内核部分有一些改动，增加了和任务切换有关的代码。

现在来看代码清单15-1。

内核的入口点在第848 行，第906 行之前的工作都和上一章相同，主要是显示处理器品牌信息，以及安装供每个任务使用的调用门。

接下来的工作是创建0 特权级的内核任务，并将当前正在执行的内核代码段划归该任务。当前代码的作用是创建其他任务，管理它们，所以称做任务管理器，或者叫程序管理器。

任务状态段（TSS）是一个任务存在的标志，没有它，就无法执行任务切换，因为任务切换时需要保存旧任务的各种状态数据。第909～911 行用于申请创建TSS 所需的内存。为了追踪程序管理器的TSS，需要保存它的基地址和选择子，保存的位置是内核数据段。第431 行，声明并初始化了6 字节的空间，前32 位用于保存TSS 的基地址，后16 位则是它的选择子。

接着，第914～918 行对TSS 进行最基本的设置。程序管理器任务没有自己的LDT，任务可以没有自己的LDT，这是允许的。程序管理器可以将自己所使用的段描述符安装在GDT 中。另外，程序管理器任务是运行在0 特权级别上的，不需要创建额外的栈。因为除了从门返回外，不能将控制从高特权级的代码段转移到低特权级的代码段。

第923～928 行，在GDT 中创建TSS 的描述符。必须创建TSS 的描述符，而且只能安装在GDT 中。

为了表明当前正在任务中执行，所要做的最后一个工作是将当前任务的TSS 选择子传送到任务寄存器TR 中。第932 行正是用来完成这个工作的。执行这条指令后，处理器用该选择子访问GDT，找到相对应的TSS 描述符，将其B 位置“1”，表示该任务正在执行中（或者处于挂起状态）。同时，还要将该描述符传送到TR 寄存器的描述符高速缓存器中。

第935、936 行，任务管理器显示一条信息：

```
    [PROGRAM MANAGER]: Hello! I am Program Manager,run at CPL=0.Now,create
user task and switch to it by the CALL instruction...
```

信息文本位于内核数据段中，代码清单的第434 行声明了标号prgman_msg1，并初始化了以上的字符串。本章后面还有其他一些字符串，也是在内核数据段声明和初始化的，不再赘述。

方括号中显示了信息的来源，是程序管理器。后面那段话的意思是“你好！我是程序管理器，运行在0 特权级上。现在，我要创建并通过CALL 指令切换到用户任务……”。

让任务之间对话，这是本章的特点，有助于更好地理解任务切换过程。既然要创建另外的任务，并执行任务切换，我们就来看看实际上是怎么做到的。

15.3 任务切换的方法

对多任务的支持是现代处理器的标志之一。为此，Intel 处理器提供了多种方法，以灵活地在各个任务之间实施切换。

尽管如此，处理器并没有提供额外的指令用于任务切换。事实上，用的都是我们熟悉的老指令和老手段，但是扩展了它们的功能，使之除了能够继续执行原有的功能外，也能用于实施任务切换操作。

第一种任务切换的方法是借助于中断，这也是现代抢占式多任务的基础。原因很简单，只要中断没有被屏蔽，它就能随时发生。特别是定时器中断，能够以准确的时间间隔发生，可以用来强制实施任务切换。毕竟，没有哪个任务愿意交出处理器控制权，也没有哪个任务能精确地把握交出控制权的时机。

我们知道，在实模式下，内存最低地址端的1KB 是中断向量表，保存着256 个中断处理过程的段地址和偏移地址。当中断发生时，处理器把中断号乘以4，作为表内索引号访问中断向量表，从相应的位置取出中断处理过程的段地址和偏移地址，并转移到那里执行。

在保护模式下，中断向量表不再使用，取而代之的，是中断描述符表。不要害怕，它和GDT、LDT 是一样的，用于保存描述符。唯一不同的地方是，它保存的是门描述符，包括中断门、陷阱门和任务门。如果你觉得这些术语太过于陌生，那就回忆一下调用门，这些门和调用门是非常类似的。当中断发生时，处理器用中断号乘以8（因为每个描述符占8 字节），作为索引访问中断描述符表，取出门描述符。门描述符中有中断处理过程的代码段选择子和段内偏移量，这和调用门是一样的。接着，转移到相应的位置去执行。

一般的中断处理可以使用中断门和陷阱门。回忆一下调用门的工作原理，它只是从任务的局部空间转移到更高特权级的全局空间去执行，本质上是一种任务内的控制转移行为。与此相同，中断门和陷阱门允许在任务内实施中断处理，转到全局空间去执行一些系统级的管理工作，本质上，也是任务内的控制转移行为。

但是，在中断发生时，如果该中断号对应的门是任务门，那么，性质就截然不同了，必须进行任务切换。即，要中断当前任务的执行，保

护当前任务的现场，并转换到另一个任务去执行。

如图15-2 所示，这是任务门（Task-Gate）描述符的格式。从图中可见，相对于其他各种描述符，任务门描述符中的多数区域没有使用，所以显得特别简单。

任务门描述符中的主要成份是任务的TSS 选择子。任务门用于在中断发生时执行任务切换，而执行任务切换时必须找到新任务的任务状态段（TSS）。所以，任务门应当指向任务的TSS。为了指向任务的TSS，只需要在任务门描述符中给出任务的TSS 选择子就可以了。

任务门描述符中的P 位指示该门是否有效，当P 位为“0”时，不允许通过此门实施任务切换；DPL 是任务门描述符的特权级，但是对因中断而发起的任务切换不起作用，处理器不按特权级施加任何保护。但是，这并不意味着DPL 字段没有用处，当以非中断的方式通过任务门实施任务切换时，它就有用了，关于这一点，你马上就会看到。

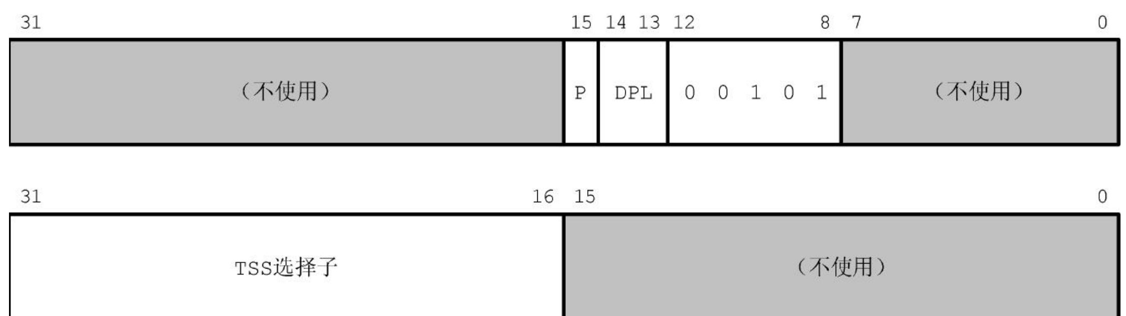


图15-2　任务门描述符的格式

这样，当中断发生时，处理器用中断号乘以8 作为索引访问中断描述符表。当它发现这是一个任务门（描述符）时，就知道应当发起任务切换。于是，它取出任务门描述符；再从任务门描述符中取出新任务的TSS 选择子；接着，再用TSS 选择子访问GDT，取出新任务的TSS 描述符。在转到新任务执行前，处理器要先把当前任务的状态保存起来。当前任务的TSS 是由任务寄存器TR 的当前内容指向的，所以，处理器把每个寄存器的“快照”保存到由TR 指向的TSS 中。然后，处理器访问新任务的TSS，从中恢复各个寄存器的内容，包括通用寄存器、标志寄存器EFLAGS、段寄存器、指令指针寄存器EIP、栈指针寄存器ESP，以及局部描述符表寄存器（LDTR）等。最终，任务寄存器TR 指向新任务的TSS，而处理器旋即开始执行新的任务。一旦新任务开始执行，处理器固件会自动将其TSS 描述符的B 位置“1”，表示该任务的状态为忙。

当中断发生时，可以执行常规的中断处理过程，也可以进行任务切换。尽管性质不同，但它们都要使用iret 指令返回。前者是返回到同一任务内的不同代码段；后者是返回到被中断的那个任务。问题是，处理器如何区分这两种截然不同的返回类型呢？

如图15-3 所示，32 位处理器的EFLAGS 有NT 位（位14），意思是嵌套任务标志（Nested Task Flag）。每个任务的TSS 中都有一个任务链接域（指向前一个任务的指针，参见上一章TSS的结构），可以填写为前一个任务的TSS 描述符选择子。如果当前任务EFLAGS 寄存器的NT 位是“1”，则表示当前正在执行的任务嵌套于其他任务内，并且能够通过TSS 任务链接域的指针返回到前一个任务。

31	21						15	14	13 12												0
保留，设置为0	ID						0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

图15-3　标志寄存器EFLAGS 的NT 位

因中断而引发任务切换时，取决于当前任务（旧任务）是否嵌套于其他任务内，其EFLAGS寄存器的NT 位可能是“0”，也可能是“1”。不过这无关紧要，因为处理器不会改变它，而是和其他寄存器一道，写入TSS 中保护起来。另外，当前任务（旧任务）肯定处于“忙”的状态，其TSS 描述符的B 位一定是“1”，在任务切换后同样保持不变。

对新任务的处理是，要把老任务的TSS 选择子填写到新任务TSS 中的任务链接域，同时，将新任务EFLAGS 寄存器的NT 位置“1”，以允许返回（转换）到前一个任务（老任务）继续执行。同时，还要把新任务TSS 描述符的B 位置“1”（忙）。

可以使用iret 指令从当前任务返回（转换）到前一个任务，前提是当前任务EFLAGS 寄存器的NT 位必须是“1”。无论任何时候处理器碰到iret 指令，它都要检查NT 位，如果此位是0，表明是一般的中断过程，按一般的中断返回处理，即，中断返回是任务内的（中断处理过程虽然属于操作系统，但属于任务的全局空间）；如果此位是1，则表明当前任务之所以能够正在执行，是因为中断了别的任务。因此，应当返回原先被中断的任务继续执行。此时，由处理器固件把当前任务EFLAGS 寄存器的NT 位改成“0”，并把TSS 描述符的B 位改成“0”（非忙）。在保存了当前任务的状态之后，接着，用新任务（被中断的任务）的TSS 恢复现场。

除了因中断引发的任务切换之外，还可以用远过程调用指令CALL，或者远跳转指令JMP 直接发起任务切换。在这两种情况下，CALL 和JMP 指令的操作数是任务的TSS 描述符选择子或任务门。以下是两个例子：

```
call 0x0010:0x00000000
jmp 0x0010:0x00000000
```

当处理器执行这两条指令时，首先用指令中给出的描述符选择子访问GDT，分析它的描述符类型。如果是一般的代码段描述符，就按普通的段间转移规则执行；如果是调用门，按调用门的规则执行；如果是TSS 描述符，或者任务门，则执行任务切换。此时，指令中给出的32 位偏移量被忽略，原因是执行任务切换时，所有处理器的状态都可以从TSS 中获得。注意，任务门描述符可以安装在中断描述符表中，也可以安装在全局描述符表（GDT）或者局部描述符表（LDT）中。

如果是用于发起任务切换，call 指令和jmp 指令也有不同之处。使用call 指令发起的任务切换类似于因中断发起的任务切换。这就是说，由call 指令发起的任务切换是嵌套的，当前任务（旧任务）TSS 描述符的B位保持原来的“1”不变，EFLAGS 寄存器的NT 位也不发生变化；新任务TSS 描述符的B 位置“1”，EFLAGS 寄存器的NT 位也置“1”，表示此任务嵌套于其他任务中。同时，TSS 任务链接域的内容改为旧任务的TSS 描述符选择子。

如图15-4 所示，假设任务1 是整个系统中的第一个任务。当任务1开始执行时，其TSS 描述符的B 位是“1”，EFLAGS 寄存器的NT 位是“0”，不嵌套于其他任务。

当从任务1 转换到任务2 后，任务1 仍然为“忙”，EFLAGS 寄存器的NT 位不变（在其TSS中）；任务2 也变为“忙”，EFLAGS 寄存器的NT位变为“1”，表示嵌套于任务1 中。同时，任务1 的TSS 描述符选择子也被复制到任务2 的TSS 中（任务链接域）。

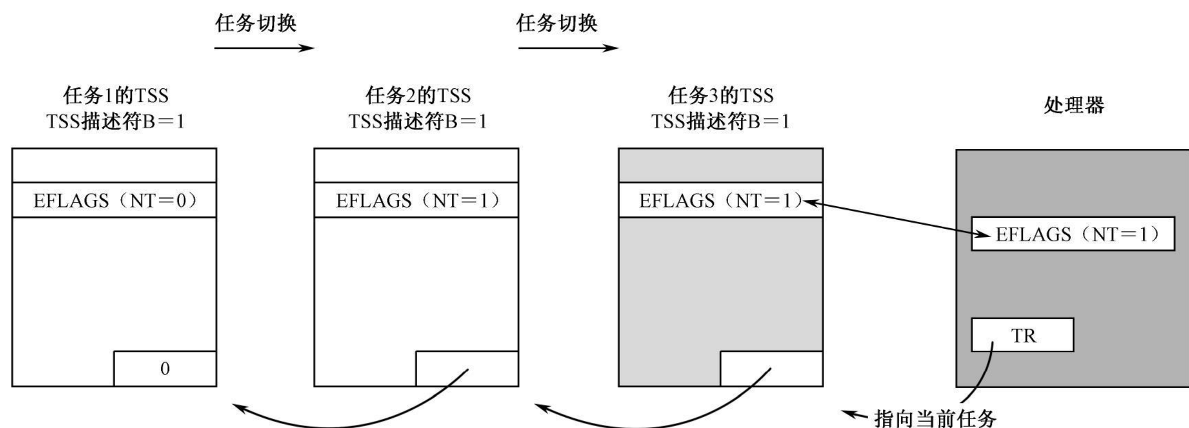


图15-4　任务嵌套示意图

最后是从任务2 转换到任务3 执行。和从前一样，任务2 保持“忙”的状态，EFLAGS 寄存器的NT 不变（在其TSS 中）；任务3 成为当前任务，其TSS 描述符的B 位变成“1”（忙），EFLAGS 寄存器的NT 位也变成“1”，同时，其TSS 的任务链接域指向任务2。

用CALL 指令发起的任务切换，可以通过iret 指令返回到前一个任务。此时，旧任务TSS 描述符的B 位，以及EFLAGS 寄存器的NT 位都恢复到“0”。

和call 指令不同，使用jmp 指令发起的任务切换，不会形成任务之间的嵌套关系。执行任务切换时，当前任务（旧任务）TSS 描述符的B 位清零，变为非忙状态，EFLAGS 寄存器的NT 位不变；新任务TSS 描述符的B 位置“1”，进入忙的状态，EFLAGS 寄存器的NT 位保持从TSS 中加载时的状态不变。

任务是不可重入的。

任务不可重入的本质是，执行任务切换时，新任务的状态不能为忙。这里有两个典型的情形：

第一种情形，执行任务切换时，新任务不能是当前任务自己。试想一下，如果允许这种情况发生，处理器该如何执行现场的保护和恢复操作？

第二种情形，如图15-4 所示，不允许使用CALL 指令从任务3 切换到任务2 和任务1 上。如果不禁止这种情况的话，任务之间的嵌套关系将会因为TSS 任务链接域的破坏而错乱。

处理器是通过TSS 描述符的B 位来检测重入的。因中断、iret、call 和jmp 指令发起任务切换时，处理器固件会检测新任务TSS 描述符的B 位，如果为“1”，则不允许执行这样的切换。

15.4　用call/jmp/iret 指令发起任务切换的实例

保护模式下的中断和异常中断处理要在第17 章才能详细阐述；和中断有关的任务切换也将在第17 章介绍。在本章，我们重点关注的是用call、jmp 和iret 指令发起的任务切换。

前面所讲的一切，都是纸上谈兵，或者说是在纸上谈任务切换。要想加深对任务切换的理解，具体的实例是必不可少的，而正在执行中的代码最能说明问题。为此，让我们回到代码清单15-1 中。

第938～945 行是用来加载用户程序的。先分配一个任务控制块（TCB），然后将它挂到TCB链上。接着，压入用户程序的起始逻辑扇区号及其TCB 基地址，作为参数调用过程load _relocate_program。

过程load_relocate_program 的工作和上一章相比没有太大变化，仅仅是对TSS 的填写比较完整。注意，这是任务切换的要求，从一个任务切换到另一个任务时，处理器要从新任务的TSS 中恢复（加载）各个寄存器的内容。尽管这是任务的第一次执行，但处理器并不知道，这是它的例行工作，你得把任务执行时，各个寄存器的内容放到TSS 中供处理器加载。

新增加的指令是从第766 行开始，到790 行结束的。首先，从栈中取出TCB 的基地址；然后，通过4GB 的内存段访问TCB，取出用户程序加载的起始地址，这也是用户程序头部的起始地址。

接着，依次登记指令指针寄存器EIP 和各个段寄存器的内容。因为这是用户程序的第一次执行，所以，TSS 中的EIP 域应该登记用户程序的入口点，CS 域应该登记用户程序入口点所在的代码段选择子。

第787～790 行，先将EFLAGS 寄存器的内容压入栈，再将其弹出到EDX 寄存器，因为不存在将标志寄存器的内容完整地传送到通用寄存器的指令。接着，把EDX 中的内容写入TSS 中EFLAGS 域。注意，这是当前任务（程序管理器）EFLAGS 寄存器的副本，新任务将使用这个副本作为初始的EFLAGS。一般来说，此时EFLAGS 寄存器的IOPL 字段为

00，将来新任务开始执行时，会用这个副本作为处理器EFLAGS 寄存器的当前值，并因此而没有足够的I/O 特权。

好，回到第947 行。

这是一条32 位间接远调用指令CALL，操作数是一个内存地址，指向任务控制块（TCB）内的0x14 单元。这样的指令我们非常熟悉，一般来说，转移到的目标位置可以由16 位的代码段选择子和32 位段内偏移量组成，也可以由16 位的调用门选择子和32 位偏移量组成。所以，从TCB 内偏移量为0x14 的地方，应当先是一个32 位的段内偏移量，接着是一个16 位的代码段或者调用门选择子。

但是，回到前一章，看图14-12，TCB 内偏移为0x14 的地方，是任务的TSS 基地址。再往后，是TSS 选择子。这很奇怪，是吗？但却是合法的。当处理器发现得到的是一个TSS 选择子，就执行任务切换。和通过调用门的控制转移一样，32 位偏移部分丢弃不用。这就是为什么我们可以把TSS 基地址作为32 位偏移量使用的原因。

当执行任务切换时，处理器用得到的选择子访问GDT，一旦它发现那是一个TSS 描述符，就知道应该执行任务切换的操作。首先，因为当前正在执行的任务是由任务寄存器TR 指示的，所以，它要把每个寄存器的“快照”保存到由TR 指向的TSS 中。

然后，处理器用指令中给出的TSS 选择子访问GDT，取得新任务的TSS 描述符，并从该TSS中恢复各个寄存器的内容，包括通用寄存器、标志寄存器EFLAGS、段寄存器、指令指针寄存器EIP、栈指针寄存器ESP，以及局部描述符表寄存器（LDTR）等。最终，任务寄存器TR 指向新任务的TSS，而处理器旋即开始执行新的任务。

谢天谢地，幸亏我们已经在load_relocate_program 过程内完整地设置了新任务的TSS，尤其是它的LDT 域、EIP 域、CS 域和DS 域，LDT 域指向用户程序的局部描述符表，EIP 域指向用户程序的入口点，CS 域指向用户程序的代码段，DS 域指向用户程序头部段。

程序管理器是计算机启动以来的第一个任务，在任务切换前，其TSS 描述符的B 位是“1”，EFLAGS 寄存器的NT 位是“0”。因为本次任务切换是用CALL 指令发起的，因此，任务切换后，其TSS 描述符的B 位仍旧是“1”，EFLAGS 寄存器的NT 位不变。当任务切换完成，用户任务成为当前任务，其TSS 描述符的B 位置“1”，表示该任务的状态为忙；EFLAGS 寄存器的NT位置“1”，表示它嵌套于程序管理器任务；TSS 的

任务链接域被修改为前一个任务（程序管理器任务）的TSS 描述符选择子。

现在，用户程序作为任务开始执行了。所以，让我们转到代码清单15-2。

总体上，用户程序的结构和上一章相比没有变化，而且功能非常简单，大部分工作都是通过调用门来完成的。程序的入口点在第55 行。

当用户任务开始执行时，段寄存器DS 指向头部段。第57、58 行，令段寄存器FS 指向头部段。其主要目的是保存指向头部段的指针以备后用，同时，腾出段寄存器DS 来完成后续操作。毕竟 ，访问数据段时，不加段超越前缀会方便很多。

第60、61 行，令段寄存器DS 指向当前任务自己的数据段。

接下来的工作是显示问候语，并报告自己的当前特权级别。因为当前特权级别是计算出来的，所以，字符串要分成两个部分显示。第63～64 行，先显示前一部分：

```
[USER TASK]: Hi! nice to meet you,I am run at CPL=
```

这句的意思是“嗨，很高兴遇到你，我运行的特权级别CPL=”。话没有说完，因为这个CPL还需要经过计算才能知道。

第66～69 行，计算当前特权级别，转换成ASCII 码后填写到数据段中，作为第二个字符串的第1 个字符。当前特权级别是由段寄存器CS 当前内容的低2 位指示的，因此，先将CS 的内容传送到AX 寄存器；接着，清除AL 寄存器的高6 位，只保留低2 位的原始内容；最后，将这个数字加上0x30，转换成可显示和打印的ASCII 码，并填写到数据段中由标号message_2 所指示的字节单元中。

第71、72 行，显示包括特权级数值在内的第二个字符串。据我们所知，当前任务的特权级别是3，因此，在屏幕上显示的完整内容是：

```
[USER TASK]: Hi! nice to meet you,I am run at CPL=3.Now,I must exit...
```

意思是，“嗨，很高兴遇到你，我运行的特权级别CPL=3。现在，必须退出喽……”。

通过在中断描述符表中安装任务门，可以在中断信号的驱使下周期性地发起任务切换。否则，每个任务都应该在适当的时候主动转换到其他任务，以免计算机的操作者发现别的任务都僵在那里没有任何反应。如果每个任务都能自觉地做到这一点，那么，这种任务切换机制被称为是协同式的。

一般来说，可以在任务内的任何地方设置一条任务切换指令，以发起任务切换。当然，如果你是为某个流行的操作系统写程序，必须听从操作系统设计者的建议，他们的软件开发指南上会告诉你怎么做。

当前任务的做法稍有些特殊，它很简单，在显示了信息之后，第74行，通过调用门转到全局空间执行。从该调用门的符号名“TerminateProgram”上看，意图是终止当前任务的执行，而不是临时转换到其他任务。

不管怎样，让我们回到代码清单15-1 中去，看看该任务在进入全局空间之后都做了些什么。

用户程序通过调用门进入任务的全局空间后， 实际的入口点在354行， 即名字为terminate_current_task 的过程。该过程用来结束当前任务的执行，并转换到其他任务。

不要忘了，我们现在仍处在用户任务中，要结束当前的用户任务，可以先切换到程序管理器任务，然后回收用户程序所占用的内存空间，并保证不再转换到该任务。为了切换到程序管理器任务，需要根据当前任务EFLAGS 寄存器的NT 位决定是采用iret 指令，还是jmp 指令。

第358～360 行，先将EFLAGS 寄存器的当前内存压栈，然后，用ESP 寄存器作为地址操作数访问栈，取得EFLAGS 的压栈值，并传送到EDX 寄存器。接着，将ESP 寄存器的内容加上4，使栈平衡，保持压入EFLAGS 寄存器前的状态。

你可能会奇怪，为什么不直接使用下面两条指令来完成以上功能：

```
pushfd
pop edx
```

的确，这两种做法的效果是一样的，之所以采用3 条指令，是因为想演示如何通过ESP 寄存器直接访问栈。在16 位模式下，不能使用SP 作为基址，所以下面的指令是错误的：

```
        mov ax,[sp]           ;错误
```

注意，使用ESP 寄存器作为指令的地址操作数时，默认使用的段寄存器是SS，即访问栈段。

第362、363 行，令段寄存器DS 指向内核数据段，以方便后面的操作。

DX 寄存器包含了标志寄存器EFLAGS 的低16 位，其中，位14 是NT 位。第365、366 行，测试DX 寄存器的位14，看NT 标志位是0 还是1，以决定采用哪种方式（iret 或者call）回到程序管理器任务。因为当前任务是嵌套在程序管理器任务内的，所以NT 位必然是“1”，应当转到标号.b1 处继续执行。

第372、373 行，也就是标号.b1 处，先显示字符串

```
        [SYSTEM CORE]: Uh...This task initiated with CALL instruction or an
exeception/ interrupt,should use IRETD instruction to switch back...
```

该字符串位于第448 行，是在内核数据段，用标号core_msg0 声明并初始化的。该字符串的内容显示，消息来源同样是系统内核，该消息的意思是“唔……该任务是用CALL 指令，或者由一个中断/异常发起的，应当使用IRETD 指令切换回去……”。

第374 行，通过iretd 指令转换到前一个任务，即程序管理器任务。执行任务切换时，当前用户任务的TSS 描述符的B 位被清零，EFLAGS 寄存器的NT 位也被清零，并被保存到它的TSS 中。

注意，在此处，我们用的是iretd，而不是iret。实际上，这是同一条指令，机器码都是CF。在16 位模式下，iret 指令的操作数默认是16 位的，要按32 位操作数执行，须加指令前缀0x66，即66 CF。为了方便，编译器创造了iretd。当在16 位模式下使用iretd 时，编译器就知道，应当加上指令前缀0x66。在32 位模式下，iret 和iretd 是相同的，下面的示例展示了它们之间的区别：

```
[bits 16]
iret            ;编译后的机器码为 CF
iretd           ;编译后的机器码为 66 CF

[bits 32]
iret            ;编译后的机器码为 CF
iretd           ;编译后的机器码为 CF
```

当程序管理器任务恢复执行时，它的所有原始状态都从TSS 中加载到处理器，包括指令指针寄存器EIP，它指向第952 行的那条指令，紧接着当初发起任务切换的那条指令。

对于刚刚被挂起的那个旧任务，如果它没有被终止执行，则可以不予理会，并在下一个适当的时机再次切换到它那里执行。不过，现在的情况是它希望自己被终止。所以，理论上，接下来的工作是回收它所占用的内存空间，并从任务控制块TCB 链上去掉，以确保不会再切换到该任务执行（当然，现在TCB 链还没有体现出自己的用处）。遗憾的是，我们并没有提供这样的代码。所以，这个任务将一直存在，一直有效，不会消失，在整个系统的运行期间可以随时切换过去。

接下来，我们再创建一个新任务，并转移到该任务执行。

第952、953 行，程序管理器先显示一条消息。标号prgman_msg2 的位置是在第439 行，位于内核数据段，在那里初始化了字符串

```
        [PROGRAM MANAGER]: I am glad to regain control.Now,create another user task and switch to it by the JMP instruction...
```

这是程序管理器在说话，方括号中的文字显示了消息的来源。该消息的大意是“我很高兴又获得了控制，现在，创建其他用户任务，并使用JMP 指令切换到它那里”。

第955～964 行，创建新的用户任务并发起任务切换。与上次相比，这次的任务切换有几个值得注意的特点。首先，可以看出，该任务也是从硬盘的50 号逻辑扇区开始加载的，就是说，它和上一个用户任务一样，来自同一个程序。这就很清楚地说明了，一个程序可以对应着多个运行中的副本，或者说多个任务。尽管如此，它们彼此却没有任何关系，在内存中的位置不同，运行状态也不一样。

其次，这次是用JMP 指令发起的任务切换，新任务不会嵌套于旧任务中。任务切换之后，程序管理器任务TSS 描述符的B 位被清零，EFLAGS 寄存器的NT 位不变；新任务TSS 描述符的B位置位，EFLAGS 寄存器的NT 位不变，保持它从TSS 加载时的状态；任务链接域的内容不变。

由于两个任务来自于同一个程序，故完成相同的工作，最终都会通过调用门进入任务的全局空间执行。而且，在执行到第365、366 行时，EFLAGS 寄存器NT 位的测试结果必定是零，即NT＝0，当前任务并未嵌套于其他任务中，于是执行第367～369 行，首先显示字符串：

```
        [SYSTEM CORE]: Uh...This task initiated with JMP instruction,  should
switch to Program Manager directly by the JMP instruction...
```

方括号内显示了消息的来源，即系统内核。该消息的意思是，“唔……该任务是用JMP 指令发起的，应当直接用JMP 指令转换到程序管理器……”。

然后，使用32 位间接远转移指令JMP 转换到程序管理器任务。指令中的标号prgman_tss 位于内核数据段（第431 行），在那里初始化了6字节，即16 位的TSS 描述符选择子和32 位的TSS基地址。按道理，这里不应该是TSS 基地址，而应当是一个32 位偏移量。不过，这是无所谓的，当处理器看到选择子部分是一个TSS 描述符选择子时，它将偏移量丢弃不用。

从第二个任务返回程序管理器任务时，执行点在第966 行。从这一行开始，一直到第969行，用于显示一条消息，然后停机。消息的内容是：

```
        [PROGRAM MANAGER]: I am gain control again, HALT...
```

消息的来源是程序管理器任务，它说：“我又获得了控制，停机……”

最后，处理器执行halt 指令，终于变消停了。

15.5　处理器在实施任务切换时的操作

处理器用以下四种方法将控制转换到其他任务：

●当前程序、任务或者过程执行一个将控制转移到GDT 内某个TSS 描述符的jmp 或者call指令；

●当前程序、任务或者过程执行一个将控制转移到GDT 或者当前LDT 内某个任务门描述符的jmp 或者call 指令；

●一个异常或者中断发生时，中断号指向中断描述表内的任务门；

●在EFLAGS 寄存器的NT 位置位的情况下，当前任务执行了一个iret 指令。

jmp、call、iret 指令或者异常和中断，是程序重定向的机制，它们所引用的TSS 描述符或者任务门，以及EFLAGS 寄存器NT 标志的状态，决定了任务切换是否，以及如何发生。

在任务切换时，处理器执行以下操作：

① 从JMP 或者CALL 指令的操作数、任务门或者当前任务的TSS 任务链接域取得新任务的TSS 描述符选择子。最后一种方法适用于以iret 发起的任务切换。

② 检查是否允许从当前任务（旧任务）切换到新任务。数据访问的特权级检查规则适用于jmp 和call 指令，当前（旧）任务的CPL 和新任务段选择子的RPL 必须在数值上小于或者等于目标TSS 或者任务门的DPL。异常、中断（除了以int n 指令引发的中断）和iret 指令引起的任务切换忽略目标任务门或者TSS 描述符的DPL。对于以int n 指令产生的中断，要检查DPL。

③ 检查新任务的TSS 描述符是否已经标记为有效（P=1），并且界限也有效（大于或者等于0x67，即十进制的103）。

④ 检查新任务是否可用，不忙（B=0，对于以CALL、JMP、异常或者中断发起的任务切换）或者忙（B=1，对于以iret 发起的任务切换）。

⑤ 检查当前任务（旧任务）和新任务的TSS，以及所有在任务切换时用到的段描述符已经安排到系统内存中。

⑥ 如果任务切换是由jmp 或者iret 发起的，处理器清除当前（旧）任务的忙（B）标志；如果是由call 指令、异常或者中断发起的，忙（B）标志保持原来的置位状态。

⑦ 如果任务切换是由iret 指令发起的，处理器建立EFLAGS 寄存器的一个临时副本并清除其NT 标志；如果是由call 指令、jmp 指令、异常或者中发起的，副本中的NT 标志不变。

⑧ 保存当前（旧）任务的状态到它的TSS 中。处理器从任务寄存器中找到当前TSS 的基地址，然后将以下寄存器的状态复制到当前TSS中：所有通用寄存器、段寄存器中的段选择子、刚才那个EFLAGS 寄存器的副本，以及指令指针寄存器EIP。

⑨ 如果任务切换是由call 指令、异常或者中断发起的，处理器把从新任务加载的EFLAGS 寄存器的NT 标志置位；如果是由iret 或者jmp 指令发起的，NT 标志位的状态对应着从新任务加载的EFLAGS 寄存器的NT 位。

⑩ 如果任务切换是由call 指令、jmp 指令、异常或者中断发起的，处理器将新任务TSS 描述符中的B 位置位；如果是由iret 指令发起的，B位保持原先的置位状态不变。

⑪ 用新任务的TSS 选择子和TSS 描述符加载任务寄存器TR。

⑫ 新任务的TSS 状态数据被加载到处理器。这包括LDTR 寄存器、PDBR（控制寄存器CR3）、EFLAGS 寄存器、EIP 寄存器、通用寄存器，以及段选择子。载入状态期间只要发生一个故障，架构状态就会被破坏（因为有些寄存器的内容已被改变，而且无法撤销和回退）。所谓架构，是指处理器对外公开的那一部分的规格和构造；所谓架构状态，是指处理器内部的各种构件，在不同的条件下，所建立起来的确定状态。当处理器处于某种状态时，再施加另一种确定的条件，可以进入另一种确定的状态，这应当是严格的、众所周知的、可预见的。否则，就意味着架构状态遭到了破坏。

⑬ 与段选择子相对应的描述符在经过验证后也被加载。与加载和验证新任务环境有关的任何错误都将破坏架构状态。注意，如果所有的检查和保护工作都已经成功实施，处理器提交任务切换。如果在从第1 步到第11 步的过程中发生了不可恢复性的错误，处理器不能完成任务切换，并确保处理器返回到执行发起任务切换的那条指令前的状态。如果在第12 步发生了不可恢复性的错误，架构状态将被破坏；如果在提交点（第13 步）之后发生了不可恢复性的错误，处理器完成任务切换并在开始执行新任务之前产生一个相应的异常。

⑭ 开始执行新任务。

在任务切换时，当前任务的状态总要被保存起来。在恢复执行时，处理器从EIP 寄存器的保存值所指向的那条指令开始执行，这个寄存器的值是在当初任务被挂起时保存的。

任务切换时，新任务的特权级别并不是从那个被挂起的任务继承来的。新任务的特权级别是由其段寄存器CS 的低2 位决定的，而该寄存器的内容取自新任务的TSS。因为每个任务都有自己独立的地址空间和任务状态段TSS，所以任务之间是彼此隔离的，只需要用特权级规则控制对TSS 的访问就行，软件不需要在任务切换时进行显式的特权级检查。

任务状态段TSS 的任务链接域和EFLAGS 寄存器的NT 位用于返回前一个任务执行，当前EFLAGS 寄存器的NT 位是“1”表明当前任务嵌套于其他任务中。无论如何，新任务的TSS 描述符的B 位都会被置位，旧任务的B 位取决于任务切换的方法。表15-1 给出了不同条件下，B 位、NT 位和任务链接域的变化情况。

表15-1　不同任务切换方法对B 位、NT 位和任务链接域的影响

标志或 TSS 任务链接域	jmp 指令的影响	call 指令或中断的影响	iret 指令的影响
新任务的 B 位	置位。原先必须为零	置位。原先必须为零	不变。原先必须被置位
旧任务的 B 位	清零	不变。原先必须是置位的	清零
新任务的 NT 标志	设置为新任务 TSS 中的对应值	置位	设置为新任务 TSS 中的对应值
旧任务的 NT 标志	不变	不变	清零
新任务的任务链接域	不变	用旧任务的 TSS 描述符选择子加载	不变
旧任务的任务链接域	不变	不变	不变

15.6　程序的编译和运行

首先，虚拟硬盘主引导扇区依然保留和上一章相同的内容。然后，编译本章中提供的两个源程序并写入虚拟硬盘。按要求，从逻辑扇区1 开始写入内核程序，从逻辑扇区50 写入用户程序。

完成后，启动虚拟机，应该可以看到图15-5 所示的画面。

```
LEARN-ASM [正在运行] - Oracle VM VirtualBox
控制(M)  View  设备(D)  帮助(H)
  If you seen this message,that means we are now in protect mode,and the system
core is loaded,and the video display routine works perfectly.

  AMD Sempron(tm) 145 Processor

  System wide CALL-GATE mounted.

[PROGRAM MANAGER]: Hello! I am Program Manager,run at CPL=0.Now,create user task
 and switch to it by the CALL instruction...

[USER TASK]: Hi! nice to meet you,I am run at CPL=3.Now,I must exit...

[SYSTEM CORE]: Uh...This task initiated with CALL instruction or an exeception/
interrupt,should use IRETD instruction to switch back...

[PROGRAM MANAGER]: I am glad to regain control.Now,create another user task and
switch to it by the JMP instruction...

[USER TASK]: Hi! nice to meet you,I am run at CPL=3.Now,I must exit...

[SYSTEM CORE]: Uh...This task initiated with JMP instruction,  should switch to
Program Manager directly by the JMP instruction...

[PROGRAM MANAGER]: I am gain control again,HALT..._
                                                              Right Ctrl
```

图15-5　本章程序运行结果

本章习题

1．修改本章的源程序，使之能够顺序完成以下工作：

① 从程序管理器任务切换到任务A，显示消息后返回程序管理器；

② 从程序管理器任务切换到任务B，显示消息后返回程序管理器；

③ 再从程序管理器任务切换到任务A，显示另一条消息，然后返回程序管理器；

④ 再从程序管理器任务切换到任务B，显示另一条消息，再返回程序管理器。

2．修改本章源程序，使之能够顺序完成以下工作：

① 从程序管理器任务切换到任务A，显示一条消息；

② 再从任务A 转换到任务B，显示一条消息；

③ 从B 直接返回到程序管理器任务。

第16章　分页机制和动态页面分配

Intel 处理器访问内存的基本策略是分段。在16 位实模式下，段的起始位置必须对齐在16 字节边界上，而且段的长度最大为64KB。

进入32 位保护模式之后，进一步强化了分段功能，并提供了保护机制。此时，段可以起始于任何位置，段的长度可以扩展到处理器的最大寻址范围边界。典型地，早期的32 位处理器拥有32根地址线，因此，段的长度可以扩展到4GB。

在32 位保护模式下，对段的访问本着“先登记，后访问”的原则进行。登记就是在GDT 或LDT 中登记段的描述符，规定了段的地址和边界，以及访问权限；访问时，则需要拿着一个段描述符的选择子才行，这就是传说中的“虎符”。处理器用段界限和特权级别来审查对段的访问，任何非法的造访行为都会被处理器阻止，并立即拉响警报，也就是所谓的异常中断。

一般来说，人们使用计算机要先安装一个操作系统。在这种情况下，段是由操作系统负责管理的。操作系统加载应用程序，根据程序的要求，为它创建一个或多个段，然后把控制权交给它。

当同时运行的程序和任务很多时，内存可能就不够用了。这时，操作系统的价值就体现出来了。每个段描述符有A 位，每当访问一个段时，处理器会将其置位。A 位的清零由操作系统定时进行，它可以借此机会统计段的访问频度。当内存不够用时，它可以将那些较少访问的段换出到磁盘上，以腾出空间来给马上要运行的段使用。一旦某个段被挪到磁盘上，操作系统应当将其描述符的P 位清零。过一段时间，当这个段又被访问时，因其描述符的P 位是“0”，处理器引发段不存在异常（中断号为11）。这类中断通常是由操作系统负责处理的，它会用同样的方法腾出空间，将这个段的内容从磁盘调入内存。当这类中断返回时，处理器会再次执行引发异常的那条指令（而不是下一条指令），于是程序又能继续执行了。

但是，因为段的长度不定，在分配内存时，可能会发生内存中的空闲区域小于要加载的段，或者空闲区域远远大于要加载的段。在前一种情况下，需要另外寻找合适的空闲区域；在后一种情况下，分配会成

功，但太过于浪费。为了解决这个问题，从80386 处理器开始，引入了分页机制。

分页功能从总体上说，是用长度固定的页来代替长度不一定的段，藉此解决因段长度不同而带来的内存空间管理问题。尽管操作系统也可以用软件来实施固定长度的内存分配，但太过于复杂，由处理器固件来做这件事，可以使速度和效率最大化。

本章的学习目标是：

1．了解页目录、页表的结构和作用，清楚为什么当我们访问一个段中的某单元时，处理器能准确地知道它在哪个页，以及页内位置的基本原理。

2．了解开启分页机制的方法和需要的准备工作。

3．了解任务的全局空间和局部空间是如何与它的页目录建立映射关系的。

4．学习按需分配页面（动态分配页面）的一般方法。

5．因为在分页机制下无法使用物理地址工作，因此，需要掌握用线性地址访问页目录表和页表，并修改目录项及页表项的手段。

6．了解什么是平坦内存模型，学习如何在平坦模型下创建程序的段描述符，知道向上扩展的数据段也能作为栈段。

7．学习用Bochs 调试分页机制下的程序。

8．学习若干新的x86 指令，包括bts、btr、btc 和bt 等。

16.1　分页机制概述

16.1.1　简单的分页模型

分段的内存管理模式是我们再熟悉不过的了，因为这是我们一贯的工作方式。如图16-1 所示，在处理器中有负责分段管理的段部件。每个程序或任务都有自己的段，这些段都用段描述符定义。随着程序的执行，当要访问内存时，就用段地址加上偏移量，段部件就会输出一个线性地址。在单纯的分段模式下，线性地址就是物理地址。

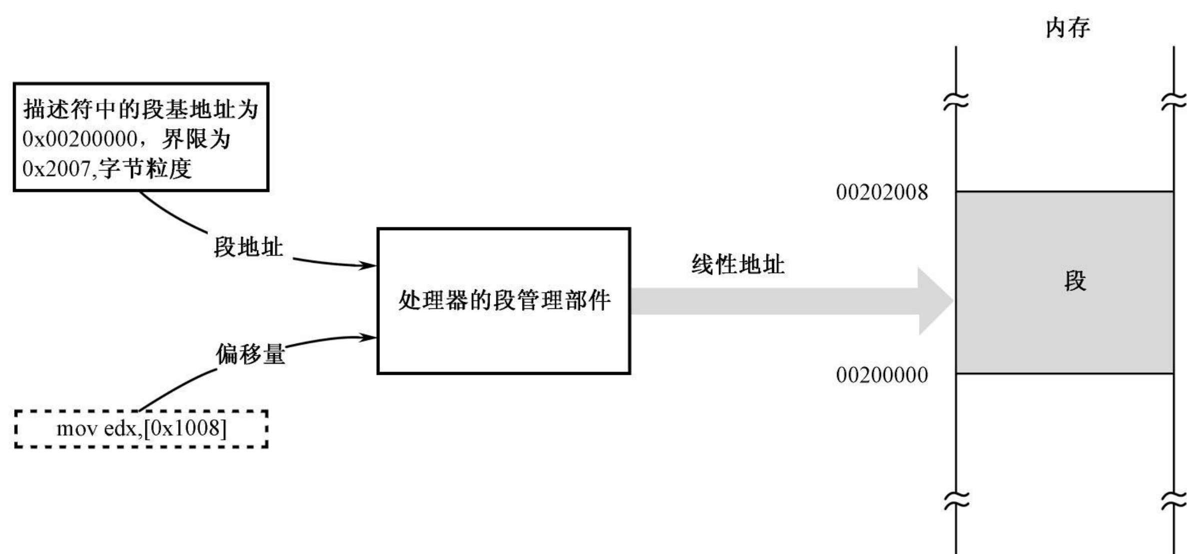


图16-1　分段机制下的线性地址就是物理地址

正如图16-1 所示，描述符中的段基地址为0x00200000，界限值为0x2007。因为段的粒度是字节，故该段的长度为8200 字节。当访问内存时，用段基地址0x00200000 加上段内偏移量0x1008，段部件就会形成线性地址0x00201008，这也是物理地址。

一旦决定采用页式内存管理，就应当把4GB 内存分成大小相同的页。但是，页在物理内存中位置是有讲究的，并不是在内存中随便找个位置，说：“来，页就从这里开始！”事实上，不是这样的。如图16-2 所示，页的最小单位是4KB，也就是4096 字节，用十六进制数表示就是0x1000。因此，第1 个页的物理地址是0x00000000，第2 个页的物理地

址是0x00001000，第3 个页的物理地址是0x00002000，……，最后一个页的物理地址是0xFFFFF000。这样，可以将4GB内存划分为1048576（0x100000）个页。很显然，页的物理地址，其低12 位始终为全零。

段管理机制对于Intel 处理器来说是最基本的，任何时候都无法关闭。也就是说，即使启用页管理功能，分段机制依然是起作用的，段部件也依然工作。

分页机制也没有增加程序员的负担，程序依然是按段来组织的。问题在于，如何将较大的段，映射到大小相同的页面上呢？

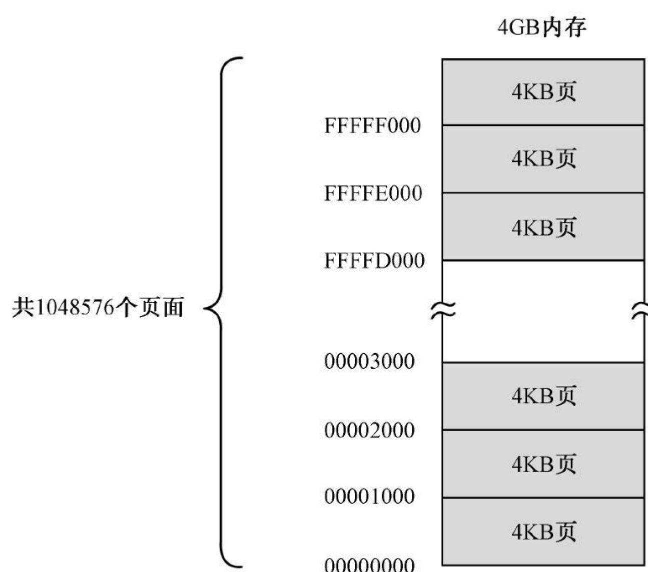


图16-2　将4GB 内存划分成以4KB 为单位的页

如图16-3 所示，内存的分配涉及段空间的分配和页分配。请仔细看这幅图，左边是虚幻的，或者说虚拟的4GB 内存空间，称为虚拟内存；右边呢，是实实在在的内存，被分成1048576 个 4KB 页面。

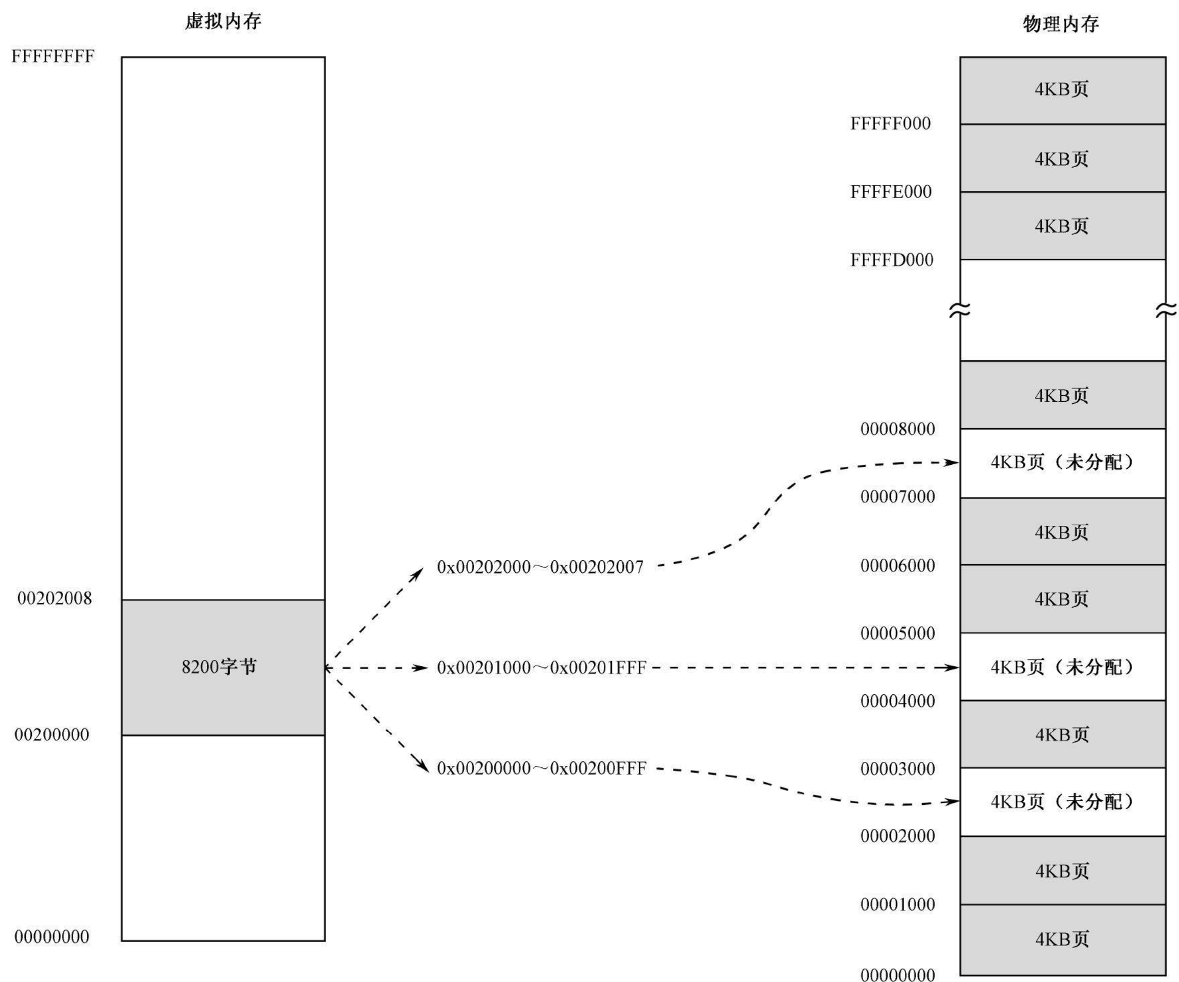


图16-3　段到页的映射

在分页模式下，操作系统可以创建一个为所有任务共用的4GB 虚拟内存空间，也可以为每一个任务创建独立的4GB 虚拟内存空间，这都是可行的。当一个程序加载时，操作系统既在要左边的虚拟内存中分配段空间，又要在右边的物理内存中分配相应的页面。因此，第一个步骤是寻找空闲的段空间，该段空间既没有被其他程序使用，也没有被同一程序内的其他段使用。比如图16-3 所示，假设已经成功找到并分配了一个段空间，基地址为0x00200000，长度为8200 字节。

页的最小尺寸是4KB，也就是4096 字节。因此，8200 字节的段，需要占用3 个页面，其中最后一个页面只用了8 个字节，其余都浪费着，但这无关紧要，如果允许页共享，多个段或多个程序可以用同一个页来存放各自的数据。

在分段之后，操作系统的任务是把段拆开，并分别映射到物理页。注意，段必须是连续的，但不要求所分配的页都是连续的、挨在一起的。事实上，在开机之后，会运行不同的程序，这都要分配页。然后，有些程序关闭了，页面要回收。几个回合下来，空闲的页零零散散地分布在物理内存中，一般不会是连续的。分配页面时，操作系统会搜索那些空闲的页，并分配给程序使用，所分配页面的总长度要大于等于段长度。

作为一个具体的例子，操作系统为程序分配了一个段，段是在虚拟内存中分配的，起始地址为0x00200000。该段有8200 字节，需要分配3个页面。为此，操作系统在物理内存中搜索可用的空闲页，还真找到了，这三个页面的物理地址分别是0x00002000、0x00004000 和0x00007000。接下来，要建立线性地址和页之间的对应关系，在图中，0x00200000～0x00200FFF 对应着物理地址为0x00002000 的页，0x00201000 ～ 0x00201FFF 对应着物理地址为0x00004000 的页，0x00202000～0x00202007 对应着物理地址为0x00007000 的页。当然，这里只是示例，线性地址区间和页的对应关系可以随意。

4GB 虚拟内存空间不可能用来保存任何数据，因为它是虚拟的，它只是用来指示内存的使用情况。当操作系统加载一个程序并创建为任务时，操作系统在虚拟内存空间寻找空闲的段，并映射到空闲的页。然后，到真正开始加载程序时，再把原本属于段的数据按页的尺寸拆开，分开写入对应的页中。

从段部件输出的是线性地址，或者叫虚拟地址。为了根据线性地址找到页的物理地址，操作系统必须维护一张表，把线性地址转换成物理地址，这是一个反过程。

如图16-4 所示，因为有1048576 个页，所以转换表也有1048576项。这是个一维表格，每个表项占4 字节，内容为页的物理地址。这个表格的用法是这样的：因为页的尺寸是4KB，故，线性地址的低12 位可用于访问页内偏移，高20 位可用于指定一个物理页。因此，把线性地址的高20 位当成索引，乘以4，作为表内偏移量，从表中取出一个双字，那就是该线性地址所对应的页的物理地址。

如图16-4 所示，如果执行指令

```
        mov edx,[0x2002]
```

那么，段部件用段地址0x00200000 加上指令中给出的偏移量0x2002，得到线性地址0x00202002。线性地址的高20 位是表格索引，即0x00202。将索引乘以4，得到0x00808，这就是表内偏移。看图，从该单元可以取出一个双字0x00007000，这就是页物理地址。

线性地址的低12 位是页内偏移量，用页物理地址加上页内偏移量，就是最终的物理内存地址。0x00007000 加上0x002，得到0x00007002，这就是实际要访问的物理内存地址。

问题在于，为什么在表内偏移量为0x00808 的地方，会恰好是页地址0x00007000，而不是其他页地址呢？问得好。当程序加载时，操作系统会首先在虚拟内存中分配段。然后，根据段需要分成多少页，来搜索空闲页面。当段较大时，要按页的尺寸分成好几个地址区段，操作系统用每个区段的首地址，取高20 位，乘以4，作为偏移量访问表格，并将分配给该区段的页的物理地址写入该表项。最后，把原本需要写入每个区段的程序数据，写到对应的页中。

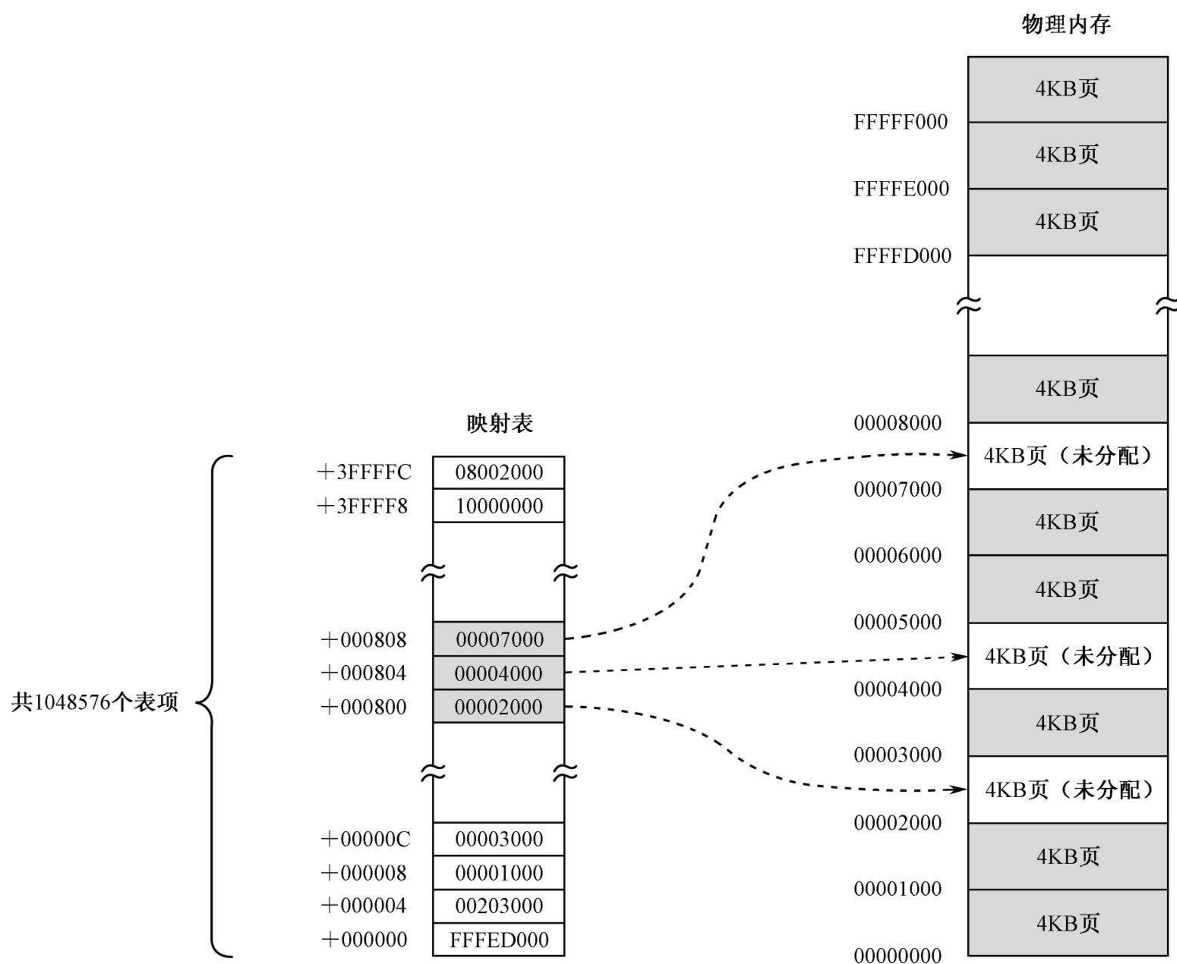


图16-4　从线性地址到页物理地址的映射

注意了，在页式内存管理中，页面的管理和分配是独立的，和分段以及段地址没有关系。操作系统所要做的，就是寻找空闲页面，把它分配给需要的段，并将页的物理地址填写到映射表内。很显然，也很重要的结论是，线性地址，包括线性地址空间，和页面分配机制没有关系。

基于以上特点，同时为了充分挖掘分页内存管理的潜力，一般来说，每个任务都可以拥有4GB 的虚拟内存空间；同时，每个任务都有自己的页映射表，如图16-5 所示。

尽管有很多任务，而且每个任务都有自己的4GB 虚拟内存空间，但是，很重要的是，在整个系统中，物理页面是统一调配的。考虑这样一种情景：任务A 有一个段，段基地址为0x00050000，段长度为3000 字节，操作系统为它分配了一个物理地址为0x08001000 的页。过了一会儿，另一个任务B 加载了，它也有一个段，段基地址也是0x00050000，

段长度为4096 字节。此时，操作系统则分配另一个不同的、物理地址为0x00700000 的页。在这种情况下，在任务A 内访问线性地址0x00050006，访问的其实是物理地址0x08001006；在任务B 内访问同样的线性地址时，访问的其实是物理地址0x00700006。

另一个会被质疑的问题是，每个任务都有4GB 虚拟内存空间，而物理内存只有一个，最大也才4GB，根本不够分的。事实上，的确不够分配。但是，操作系统可以将暂时不用的页退避到磁盘，调入马上就要使用的页，通过这种手段来实现分页内存管理。这就是为什么内存容量较小时，程序越来越慢，硬盘工作指示灯不停地闪烁的原因。

以上，就是基本的段页式内存管理机制。

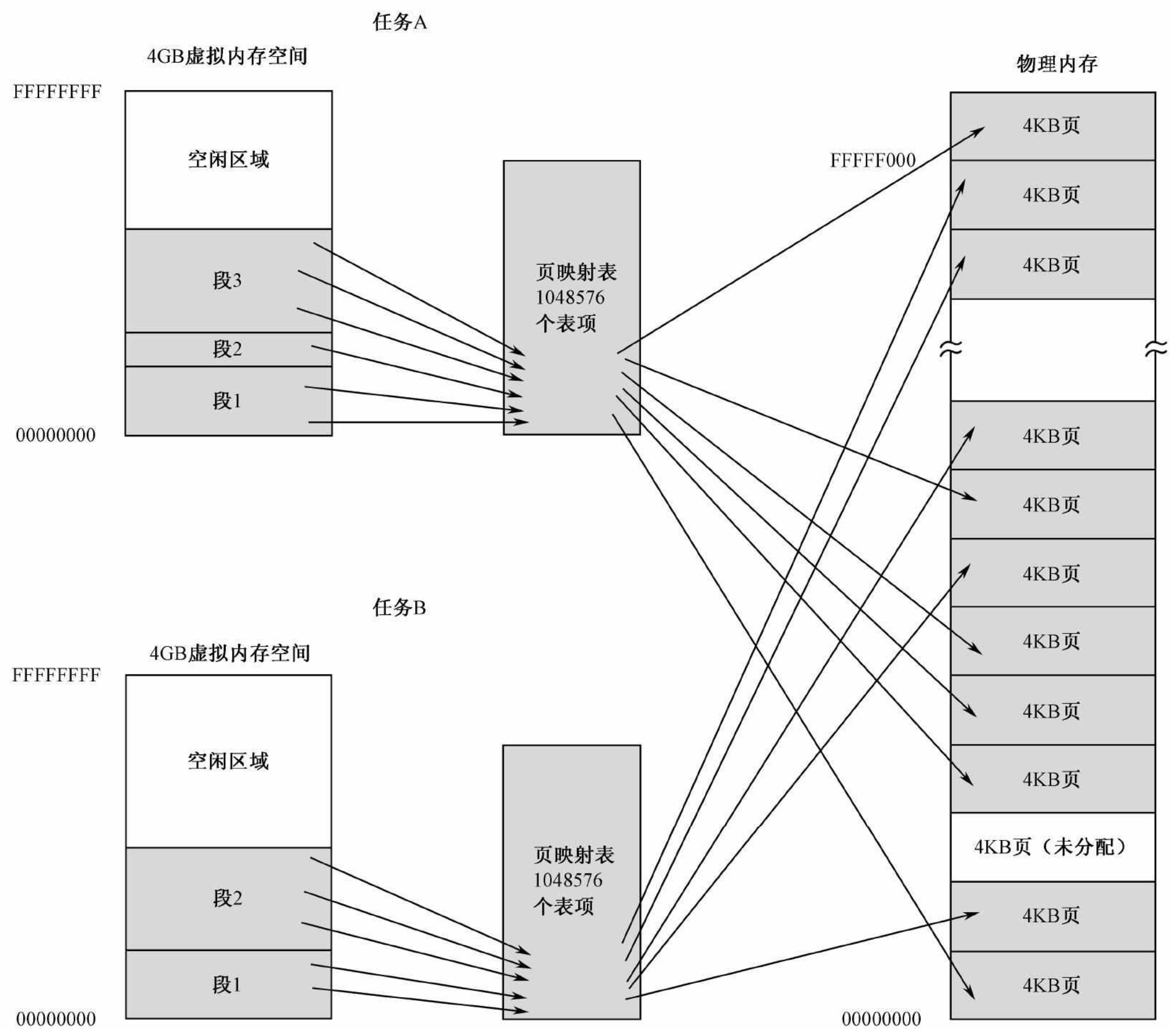


图16-5　基本的段页式内存管理示意图

16.1.2 页目录、页表和页

第一个支持分页内存管理模式的Intel 处理器是80386，那个时候，分页机制是很简单的。几十年弹指一挥间，处理器变得更为强大，而分页机制也变得复杂。任何事物都具有惯性，更何况IT 界的人常说，软件的发展速度远远跟不上硬件的前进步伐。说得好，虽然分页机制在最新的处理器上并非那么简单，但新机制毕竟用得很少，倒是早先的分页机制依然那么流行。另一方面，因为兼容方面的原因，最初的分页机制是所有处理器都支持的，从最初的分页机制开始学习，可以很容易进一步理解其他分页机制，毕竟它们是一脉相承的。

我们知道，为了完成从虚拟地址（线性地址）到物理地址的转换，操作系统应当为每个任务准备一张页映射表。因为任务的虚拟地址空间为4GB，可以分出1048576 个页，所以，映射表需要1048576 个表项，用于存放页的物理地址。又因为每个表项占4 字节，所以，映射表的总大小为4MB。

没错，这张表很大，要占用相当一部分内存空间，考虑到在实践中，没有哪个任务会真的用到所有表项，充其量只是很小一部分，这就很浪费了。

当然，你可能会建议先划出一小块内存给它，然后，根据需要再动态扩展。的确，这是可行的。但是，因为一个特殊的原因，这张表在实际使用的时候，它的前半部分和后半部分会被同时用到。具体是什么原因，马上就要讲到，也正是因为这个尚未说明的原因，这张表从一开始就必须完全定义，而且不可避免地要占用4MB 内存空间。为了解决这个问题，同时又不会浪费宝贵的内存空间，处理器设计了层次化的分页结构。

分页结构层次化的主要手段是不采用单一的映射表，取而代之的是页目录表和页表。如图16-6所示，首先，因为4GB 的虚拟内存空间对应着1048576 个4KB 的页，可以随机地抽取这些页，将它们组织在1024 个页表内，每个页表可以容纳1024 个页。页表内的每个项目叫做页表项，占4 字节，存放的是页的物理地址，故每个页表的大小是4KB，正好是一个标准页的长度。

注意，页在页表内的分布是随机的，哪个页位于哪个页表中，这是没有规律的。在一个真实的系统中，老任务不断被关闭，新任务不断被

创建并投入运行，页面的回收和再分配没有什么规律可言。

由于页表中存放的是页的物理地址，故每个页表项占4 字节，这样，每个页表占4096 字节，正好是一个物理页的大小，可以很方便地用一个物理页来定义每个页表。

如图16-6 所示，在将1048576 个页归拢到1024 个页表之后，接着，再用一个表来指向1024个页表，这就是页目录表（Page Directory Table，PDT），和页表一样，页目录项的长度为4 字节，填写的是页表的物理地址，共指向1024 个表页，所以页目录表的大小是4KB，正好是一个标准页的长度。

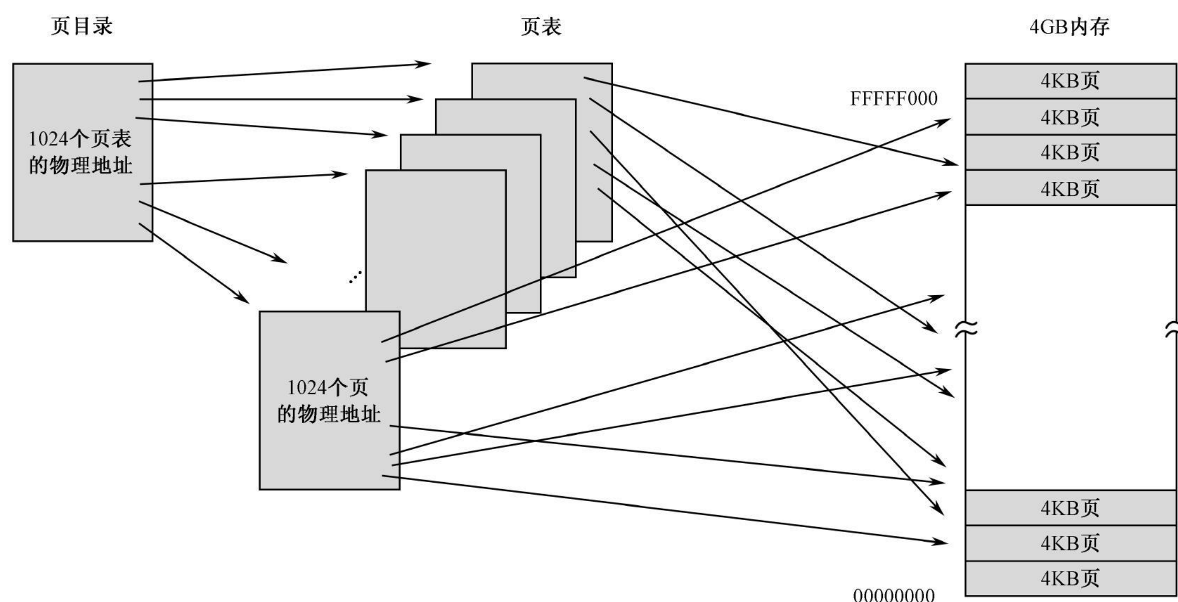


图16-6　页目录、页表和页的对应关系

这样的层次化分页结构是每个任务都拥有的，或者说，每个任务都有自己的页目录和页表。如图16-7 所示，在处理器内部，有一个控制寄存器CR3，存放着当前任务页目录的物理地址，故又叫做页目录基址寄存器（Page Directory Base Register，PDBR）。

每个任务都有自己的任务状态段（TSS），它是任务的标志性结构，存放了和任务相关的各种数据，其中就包括了CR3 寄存器域，存放了任务自己的页目录物理地址。当任务切换时，处理器切换到新任务开始执行，而CR3 寄存器的内容也被更新，以指向新任务的页目录位置。相应地，页目录又指向一个个的页表，这就使得每个任务都只在自己的地址空间内运行。

从图16-7 中还可以看出，页目录和页表也是普通的页，混迹于全部的物理页中。它们和普通页的不同之处仅仅在于功能不一样。当任务撤销之后，它们和任务所占用的普通页一样会被回收，并分配给其他任务。

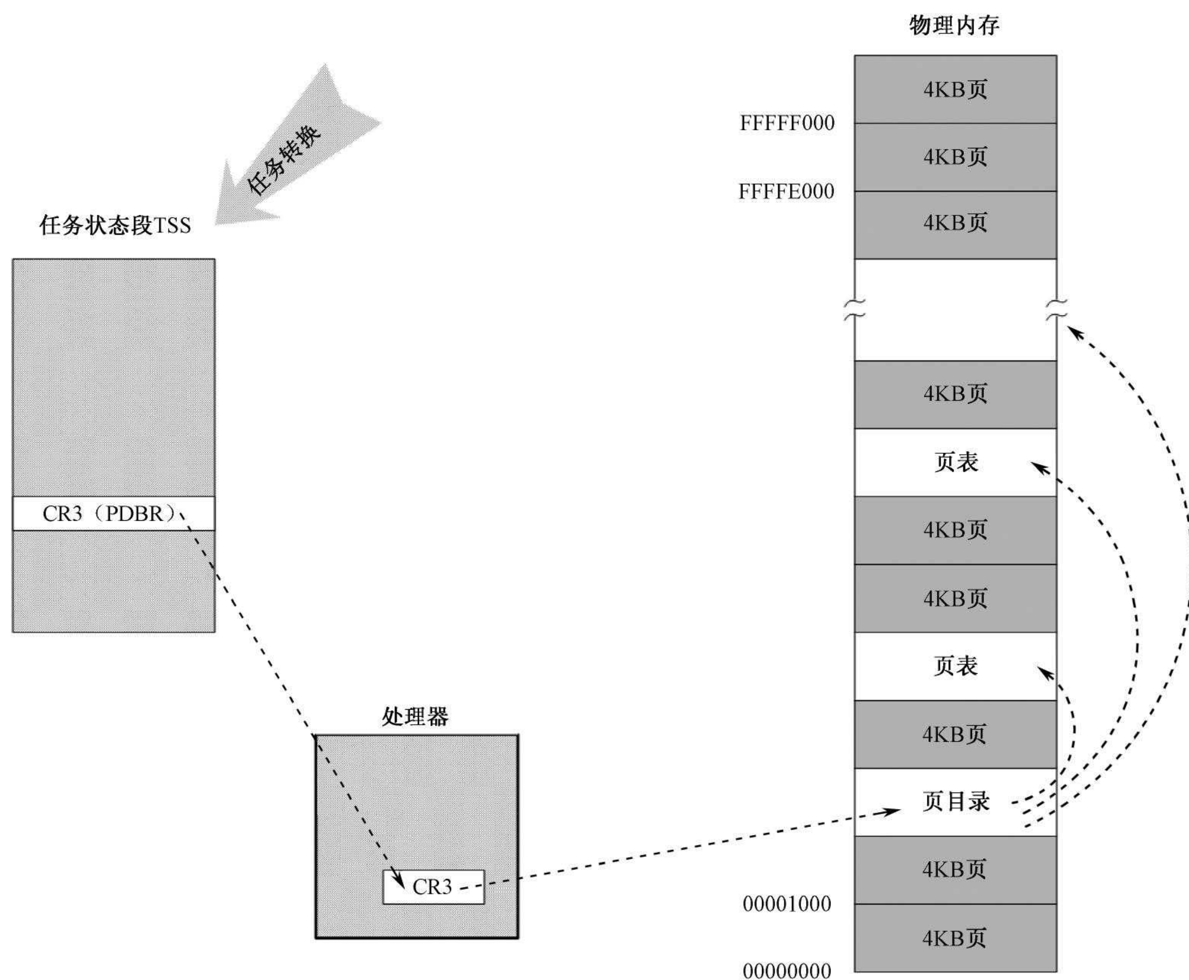


图16-7　整个分页系统的全局视图

16.1.3　地址变换的具体过程

对于Intel 处理器来说，有关分页，最简单和最基本的机制就是这些：CR3 寄存器给出了页目录的物理基地址；页目录给出了所有页表的物理地址，而每个页表给出了它所包含的页的物理地址。好了，该清楚的都清楚了，唯一还不明白的，应该是如何用这种层次性的分页结构把线性地址转换成物理地址？

这里有个例子。

假如某个任务加载后，操作系统根据它的实际情况，在其4GB 虚拟地址空间里创建了一个段，段的起始地址为0x00800000，段界限值为0x5000，字节粒度。当该任务执行时，段寄存器DS 指向该段。又假设执行了下面一条指令：

```
mov edx,[0x1050]
```

此时，段部件会输出线性地址0x00801050。在没有开启分页机制时，这就是要访问的物理内存地址，但现在开启了分页机制，所以，这是一个虚拟地址，要经过页部件的转换，才能得到物理地址。

如图16-8 所示，处理器的页部件专门负责线性地址到物理地址的转换工作。它首先将段部件送来的32 位线性地址截成3 段，分别是高10 位、中间的10 位和低12 位。高10 位是页目录的索引，中间10 位是页表的索引，低12 位则作为页内偏移来用。

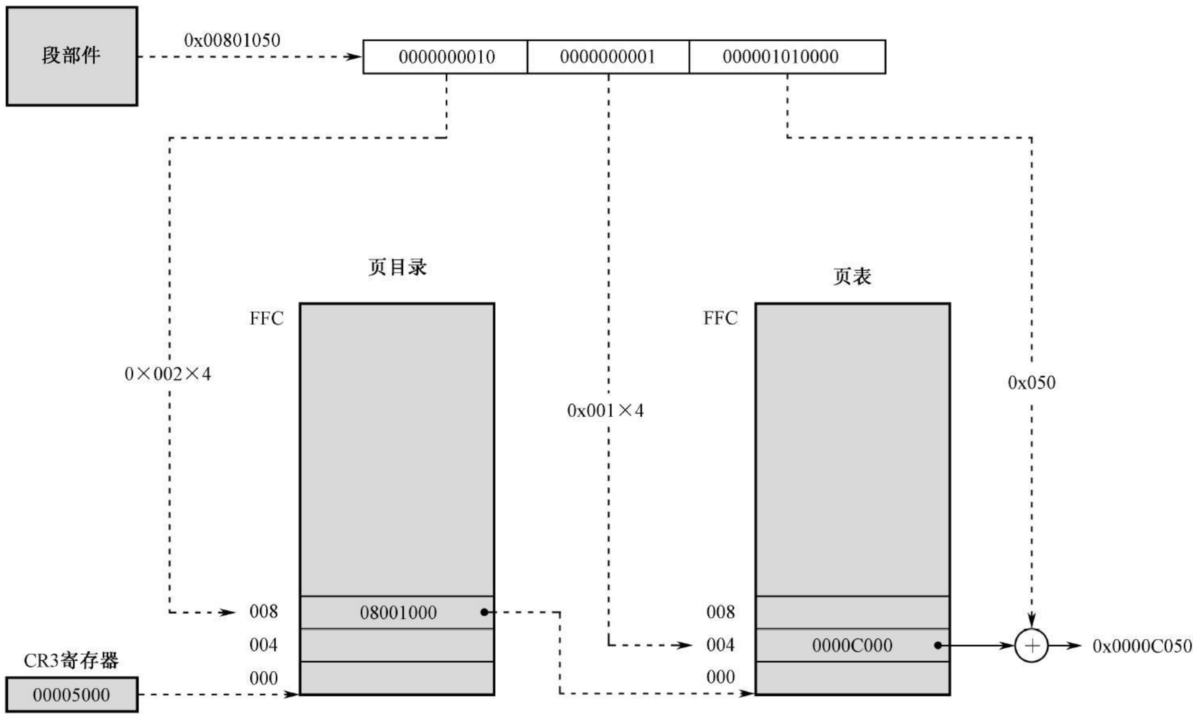


图16-8　页部件把线性地址转换为物理地址的例子

当前任务页目录的物理地址在处理器的CR3 寄存器中，假设它的内容为0x00005000。段管理部件输出的线性地址是0x00801050，其二进制的形式为0000 0000 1000 0000 0001 0000 0101 0000。高10 位为0000000010，也就是十六进制的0x002，它是页目录表内的索引，处理

器将它乘以4（因为每个目录项为4 字节），作为偏移量访问页目录。最终，处理器从物理地址00005008 处取得页表的物理地址0x08001000。

线性地址的中间10 位为二进制的0000000001，即0x001，处理器要用它作为页表内的索引来取得页的物理地址。处理器将该索引值乘以4，作为偏移量访问页表。最终，处理器又从物理地址08001004 处取得页的物理地址，这就是我们一直努力寻找的那个页。

页的物理地址是0x0000C000，而线性地址的低12 位是数据所在的页内偏移量。故处理器将它们相加，得到物理地址0x0000C050，这就是线性地址0x00801050 所对应的物理地址，要访问的数据就在这里。

注意，这种变换不是无缘无故的，而是事先安排好的。当任务加载时，操作系统先创建虚拟的段，并根据段地址的高20 位决定它要用到哪些页目录项和页表项。然后，寻找空闲的页，将原本应该写入段中的数据写到一个或者多个页中，并将页的物理地址填写到相应的页表项中。只有这样做了，当程序运行的时候，才能以相反的顺序进行地址变换，并找到正确的数据。

检测点16.1

在分页模式下，某程序运行时，段部件发出一个线性地址0x0C005032 访问内存数据。如果该线性地址对应的物理页是0x0000A000，页表的物理地址是0x00003000，那么，操作系统在此程序开始运行前，是如何安排与该线性地址相关的页目录项和页表项的？

16.2　本章代码清单

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：16-1（保护模式微型核心程序），源程序文件：c16_core.asm

本章代码清单：16-2（动态加载的用户程序），源程序文件：c16.asm

16.3　使内核在分页机制下工作

16.3.1　创建内核的页目录表和页表

必须说明的是，必须在保护模式下才能启动页功能。和往常一样，首先进入保护模式执行的是内核程序，而且，我们要先让内核在分页机制下工作。

本章依然没有提供新的主引导程序，这意味着，还要用以前的主引导程序，同时还意味着，内核程序的总体结构没有变化，否则主引导程序又怎么可能按往常的方式加载它呢。

内核的入口点是在代码清单16-1 的第884 行，即标号“start”处。执行到这里的时候，主引导程序已经创建了内核的大部分要素：全局描述符表（GDT）、公共例程段、内核数据段、内核代码段、内核栈，还包括一个用于访问全部4GB 内存空间的段。

内核的总体结构和它在内存中的布局，从第14 章以来就没什么变化，而且第14 章还曾经给出了一幅内核的内存布局图。为了方便，这里用另一种形式再次展示一下，如图16-9 所示。

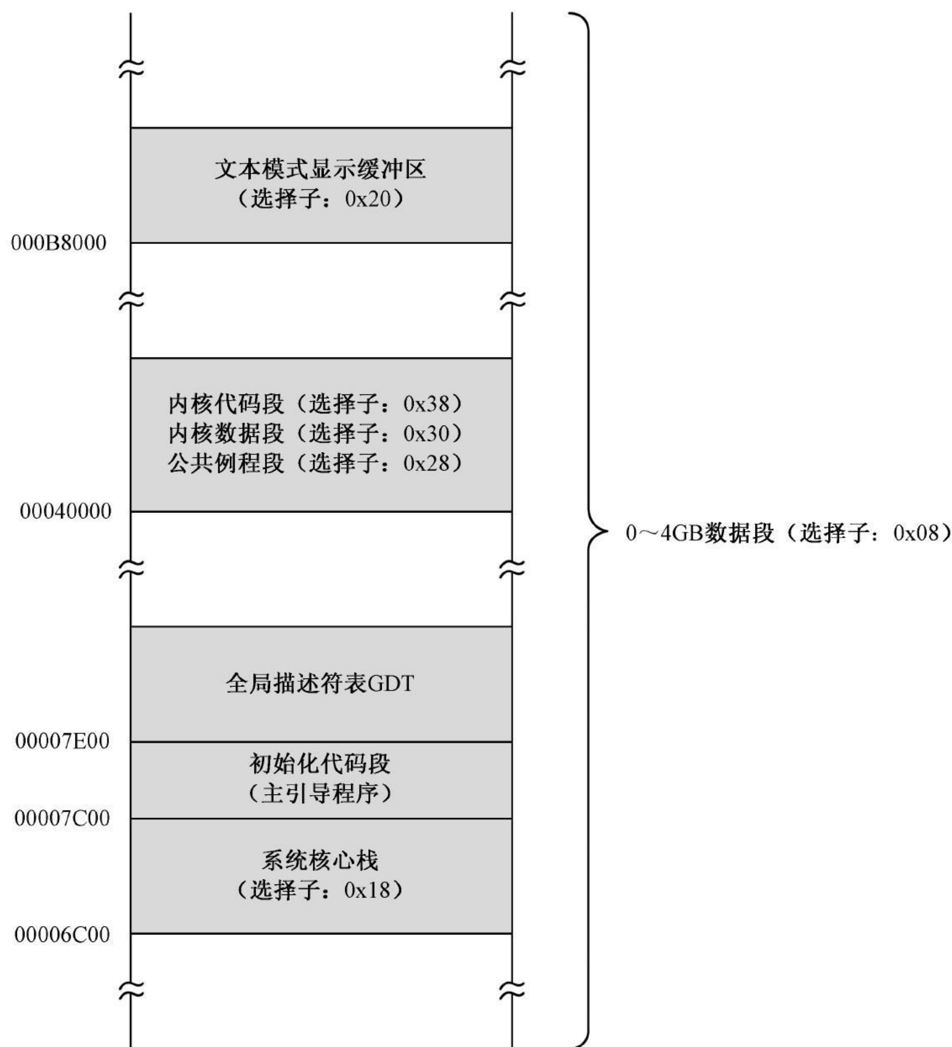


图16-9　内核加载之后的内存布局图

其中，各个段在内存中的位置、段描述符和描述符的选择子，都没有变化，可以和前面的章节对照一下。强调这些，只是要表明，即使是在分页机制下工作，对以往的代码和内存分配都没有什么影响。

接着回到代码清单16-1 中来。

第885～889 行，令段寄存器DS 和ES 分别指向内核数据段与0～4GB 数据段，以方便后面的操作。

第891、892 行，在屏幕上显示第一个字符串，表明当前正在内核中执行，而且是在保护模式下工作。

第895～921 行，在屏幕上显示处理器的品牌信息，这段代码和往常一样，没有任何变化。

接下来的工作是准备开启页功能，首先必须创建页目录和页表。每个任务都有自己的页目录和页表，内核也不例外，尽管它是为所有任务所共有的，但也包括作为任务而独立存在的部分，以执行必要的系统管理工作。因此，要想内核正常运行，必须创建它自己的页目录和页表。

麻烦在于，内核已经加载完毕，它的所有部分都已经位于内存中。当然，你可能会问，这怎么会是个麻烦事呢？原因是，在一个理想的分页系统中，要加载程序，必须先搜索可用的页，并将它们与段对应起来。在这种情况下，段部件输出的线性地址和页部件输出的物理地址不同，是很自然的事，因为一切都发生在程序加载完毕、段和页已经有了确定的映射关系之后。在这种情况下，页功能开启之后，各方都能很好地适应。

然而，由于内核是在开启页功能之前加载的，段在内存中的位置已经固定。在这种情况下，即使开启了页功能，线性地址也必须和物理地址相同才行。比如， 在开启页功能之前，GDT 在内存中的基地址是0x00007E00，它就是全局描述符表的物理地址，段部件输出的线性地址就是物理地址。在开启页功能之后，它还在那个内存位置，这就要求页部件输出的物理地址和段部件输出的线性地址相同。一句话，要求线性地址等于物理地址才行。

注意，进入分页模式之后，所有东西的地址都成了线性地址，包括GDT、LDT 和TSS 的地址，等等。

其实这也好办。

不像流行的操作系统，我们的内核非常小，这是没有办法的事，我们的任务不是写一个操作系统，能说明问题即可。也正是因为我们的内核很小，所以低端1MB 的空间对它来说已经绰绰有余了。如此一来，我们只需要将低端1MB 内存特殊处理，使这一部分内存的线性地址和经过页部件转换之后的物理地址相同即可。这样做的好处是，内核不用做任何变动即可在分页机制下正常工作。

对页目录和页表在内存中的位置没有什么限制，在哪里都行，前提是属于有效的可用内存范围，如果只安装了1GB 的物理内存，而想把页目录放到2GB 的位置，是不行的。而且，页目录和页表必须各自占用一个自然页，也就是说，它们的物理地址的低12 位必须全是零。

在页目录中，一个目录项对应着一个页表，而一个页表可以容纳1024 个页，也就是4MB 内存。所以，对于内核来说，只需要一个页表就

行了，还用不完。这就是说，一个页目录和一个页表就足够了。

再来看图16-9，在GDT和内核加载的区域之间，是一片空白。因此，我们可以将内核的页目录表放在物理地址0x00020000 处；而把内核的第一个页表放在物理地址 0x00021000 处。此时，新的低端1MB 内存布局就如图16-10所示了。

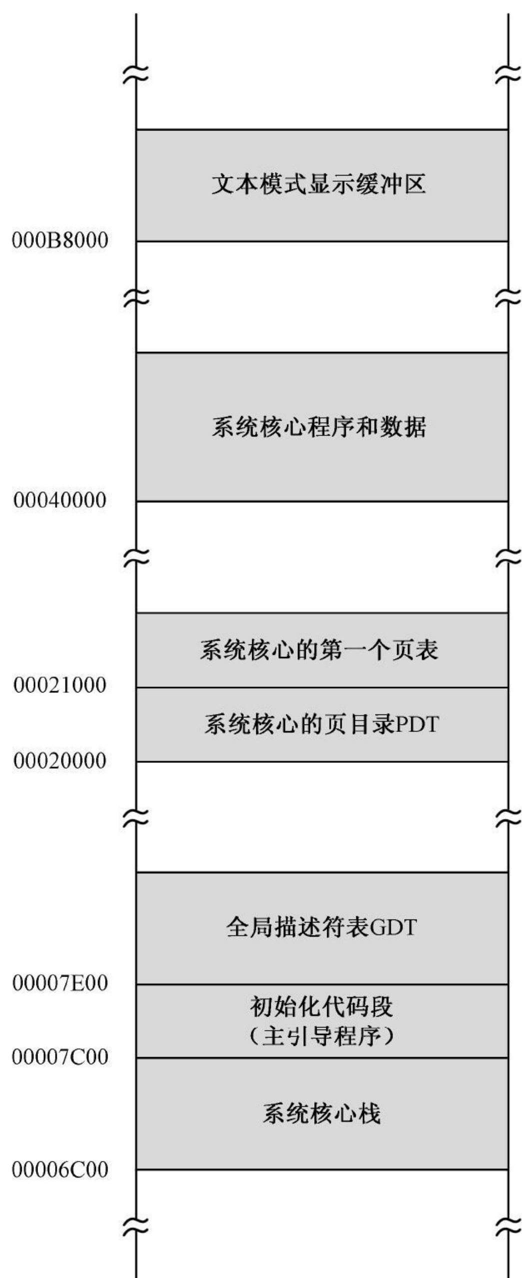


图16-10　加入页目录和页表后的低端1MB 内存布局

既然我们的目的清楚了，也知道该怎么干，那么，回到代码清单16-1，先来创建页目录表和页表。

第927～933 行，访问段寄存器ES 所指向的4GB 数据段，用0x00020000 作为偏移量，访问页目录，将所有目录项清零。

如图16-11 所示，这是页目录项和页表项的格式。可以看出，在页目录和页表中，只保存了页表或者页物理地址的高20 位。原因很简单，页表或者页的物理地址，都要求必须是4KB 对齐的，以便于放在一个页内，故其低12 位全是零。在这种情况下，可以只关心其高20 位，低12 位安排其他用途。

31～12	11～9	8	7	6	5	4	3	2	1	0
页表物理基地址的31～12	AVL	G	0	D	A	PCD	PWT	US	RW	P

（a）页目录项

31～12	11～9	8	7	6	5	4	3	2	1	0
页物理基地址的31～12	AVL	G	PAT	D	A	PCD	PWT	US	RW	P

（b）页表项

图16-11　页目录项和页表项的组成

●P（Present）是存在位，为“1”时，表示页表或者页位于内存中。否则，表示页表或者页不在内存中，必须先予以创建，或者从磁盘调入内存后方可使用。

●RW（Read/Write）是读/写位。为“0”时表示这样的页只能读取，为“1”时，可读可写。

●US（User/Supervisor）是用户/管理位。为“1”时，允许所有特权级别的程序访问；为“0”时，只允许特权级别为0、1 和2 的程序访问，特权级别为3 的程序不能访问。

● PWT（Page-level Write-Through）是页级通写位，和高速缓存有关。“通写”是处理器高速缓存的一种工作方式，这一位用来间接决定是否采用此种方式来改善页面的访问效率。由于高速缓存的知识将在下册中介绍，所以在本章中该位直接清零。

● PCD（Page-level Cache Disable）是页级高速缓存禁止位，用来间接决定该表项所指向的那个页是否使用高速缓存策略。同样，在本章中，该位将被清零；

● A（Accessed）是访问位。该位由处理器固件设置，用来指示此表项所指向的页是否被访问过。这一位很有用，可以被操作系统用来监视页的使用频率，当内存空间紧张时，用以将较少使用的页换出到磁盘，同时将其P 位清零。然后，将释放的页分配给马上就要运行的程序，以实现虚拟内存管理功能。

● D（Dirty）是脏位。该位由处理器固件设置，用来指示此表项所指向的页是否写过数据。

● PAT（Page Attribute Table）页属性表支持位。此位涉及更复杂的分页系统，和页高速缓存有关，可以不予理会，在普通的4KB 分页机制

中，处理器建议将其置“0”。

●G（Global）是全局位。用来指示该表项所指向的页是否为全局性质的。如果页是全局的，那么，它将在高速缓存中一直保存（也就意味着地址转换速度会很快）。因为页高速缓存容量有限，只能存放频繁使用的那些表项。而且，当因任务切换等原因改变CR3 寄存器的内容时，整个页高速缓存的内容都会刷新。

●AVL 位被处理器忽略，软件可以使用。

回到代码清单16-1 中来。

将页目录清零的原因，主要是使所有目录项的P 位为“0”。目录项用于定位对应的页表，如果其P 位是“0”，表明该页表并不在内存中，在地址变换时将引发处理器异常中断。

在建立了一个为空的页目录表之后，第936 行，将页目录表的物理地址登记在它自己的最后一个目录项内。页目录最大4KB，最后一个目录项的偏移量是0xFFC，即十进制数4092。页目录需要频繁地进行修改，为了方便用线性地址访问页目录表自身，需要使用这项技术，马上我们就要讲到。注意，填写的内容是0x00020003，该数值的前20 位是物理地址的高20 位；P＝1，页是位于内存中的；RW＝1，该目录项指向的页表可读可写。还要注意到，US 位是“0”，故此目录项指向的页表不允许特权级为3 的程序和任务访问。

注意，这将浪费一个页目录表项，同时使得最高端的4MB 内存无法访问（0xFFC00000～0xFFFFFFFF）。不过，即使不浪费，一般的软件也不会涉足这个区域。

如图16-12 所示，内核占用着内存的低端1MB，线性地址范围是0x00000000～0x000FFFFF，共256 个4KB 页，占用了页目录表的第1个目录项，以及该目录项下属页表的前256 个页表项。第939 行，修改页目录内第1 个目录项的内容，使其指向页表，页表的物理地址是0x00021000，该页位于内存中，可读可写，但不允许特权级别为3 的程序和任务访问。

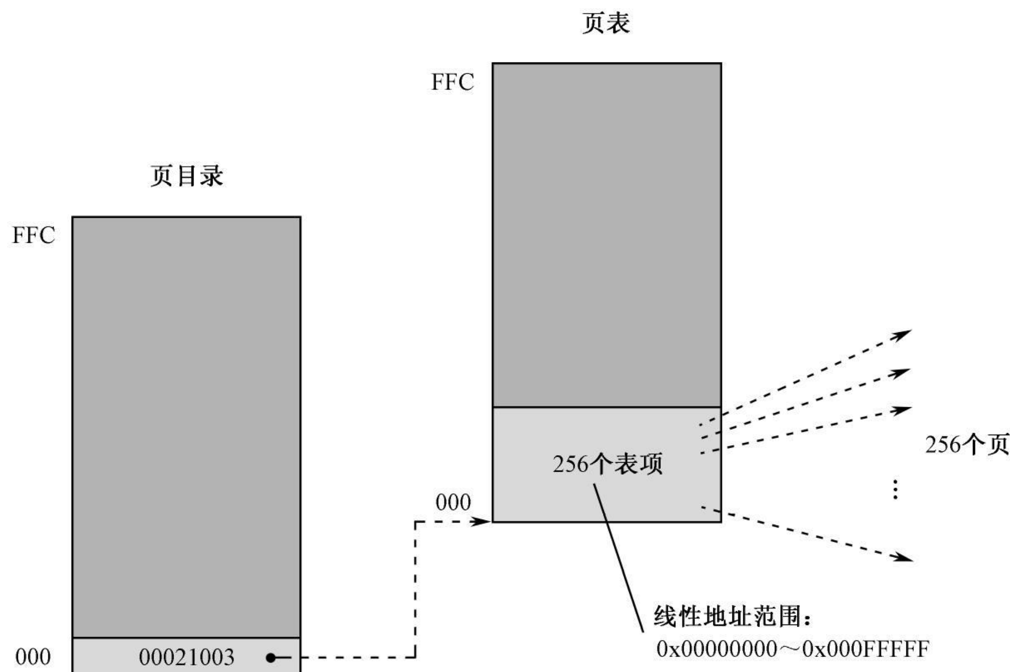


图16-12　内核所占用的低端1MB 内存分页效果图

第942～952 行，将内存低端1MB 所包含的那些页的物理地址按顺序一个一个地填写到页表中，当然，仅填写256 个页表项。第1 个页表项对应的是线性地址0x00000000～0x00000FFF，填写的内容是第1 个页的物理地址0x00000000；第2 个页表项对应的是线性地址0x00001000～0x00001FFF，填写的是第2 个页的物理地址0x00001000；第3 个页表项对应的是线性地址0x00002000～0x00002FFF，填写的是第3 个页的物理地址0x00002000，……。如此一来，这部分内存的线性地址就和物理地址一样了。

这部分代码还是很容易看懂的，第942～944 行，用EBX 寄存器指向页表基地址；用EAX 寄存器保存页的物理地址，初始为0x00000000，每次按0x1000 递增，以指向下一个页；ESI 寄存器用于定位每一个页表项。

参见图16-11，因为页的物理地址是4KB 对齐的，故其低12 位全为零，在写入页表项时，仅保存它的前20 位，低12 位是页属性。在实际写入每个页表项之前，先将页的物理地址转存到EDX 寄存器，并将属性值加到其低12 位上。属性值是3，故P＝1，RW＝1；US＝0，特权级别为3的程序和任务不能访问这些页。

尤其注意第948 行的指令：

```
        mov [es:ebx+esi*4],edx
```

再重复一次，请务必注意，32 位处理器允许在寻址时使用一个倍率因子，在这里是乘以4，表达式的计算不在编译期间进行，而在指令执行的时候进行。

页表的前256 个表项填写之后，EBX 寄存器的当前值是256，它又被用于第955～958 行，接着处理其余的表项，使它们的内容为全零。即，将它们置为无效表项。

页目录和页表都已创建，它们的表项也都安排妥当，第961、962 行，将页目录表的物理基地址传送到控制寄存器CR3，也就是页目录表基地址寄存器PDBR，该寄存器的格式如图16-13所示。

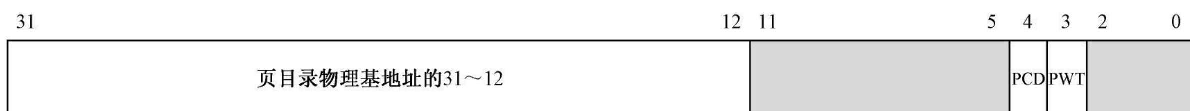


图16-13　控制寄存器CR3（PDBR）的组成

由于页目录表必须位于一个自然页内，故其物理基地址的低12 位是全零，处理器的设计者认为既然如此，只登记它的高20 位即可。低12 位，除了PCD 和PWT 位外，都没有使用。这两位用于控制页目录的高速缓存特性，请参照前面的解释。在本章中，为了方便，这两位一律为“0”。

从表面上看，和控制寄存器有关的传送指令和普通的传送指令一样。实际上，这是两种不同类型的指令，操作码是不一样的。控制寄存器是在有了32 位处理器之后才开始出现的，故其长度至少是32 位。在32 位处理器上，和控制寄存器有关的传送指令，其格式为：

```
        mov CR0～CR7,r32          ;从 32 位通用寄存器传送到控制寄存器
        mov r32,CR0～CR7          ;从控制寄存器传送到 32 位通用寄存器
```

没错，最新的处理器内共有8 个控制寄存器，从CR0 到CR7，至于它们都有什么用，别好奇，等看完这本书后，你再慢慢学习吧。汇编语言的一个缺点是无法区分不同指令间的细微差别。在这里，尽管也使用了助记符号“mov”，但实际上，它和一般的传送指令有所区别。

看来全都准备停当了，现在就开启页功能。如图16-14 所示，控制寄存器CR0 的最高位，也就是位31，是PG（Page）位，用于开启或者关

闭页功能。当该位清零时，页功能被关闭，从段部件来的线性地址就是物理地址；当它置位时，页功能开启。只能在保护模式下才能开启页功能，当PE 位清零时（实模式），设置PG 位将导致处理器产生一个异常中断。



图16-14　控制寄存器CR0 的PG 位

第964～966 行，先读取控制寄存器CR0 的原始内容，然后，将其最高位置“1”，其他各位保持原来的数值不变。接着，将修改后的内容重新传回CR0 寄存器，这直接导致处理器工作在分页机制下。从这一瞬间开始，段部件产生的地址就不再被看成物理地址，而是要送往页部件进行变换，以得到真正的物理地址。

注意，现在内核工作在分页机制的一个特殊情况下，即，线性地址和经过页部件转换后的物理地址相同，这是精心安排后的结果。举个例子，如果要访问全局描述符表GDT 内的第2 个描述符。在开启页功能之前，GDT 的线性地址是0x00007E00，第2 个描述符的线性地址则是0x00007E08。在开启页功能之后，依然要保证转换后的物理地址和线性地址一样，仍是0x00007E00 和0x00007E08。好，线性地址送到页部件，页部件用线性地址的高10 位在页目录中查找页表；再用线性地址的中间10 位在页表中查找页。经过转换，找到了包含该数据的页，页的物理地址是0x00007000。于是，将页地址和线性地址的低12 位（0xE08）拼凑在一起，形成最终的0x00007E00 和0x00007E08。

可以在Bochs 中用“creg”命令察看控制寄存器CR0 和CR3 的内容，具体方法请参见本章16.6.2 节；也可以输入一个线性地址，来察看它所对应的物理页，具体方法请参见本章16.6.3 节；要察看当前页表中的全部内容，可以用“info tab”命令，请参见本章16.6.4 节。

16.3.2　任务全局空间和局部空间的页面映射

和往常一样，接下来的工作是加载用户程序，并创建一个任务。

每个任务都有自己独立的4GB 虚拟地址空间。这话说来简单，大家也都能在理论的层面上理解，但从来没有实现过，今天我们就来实践一回。

但是细一琢磨，这里面有个问题。

每个任务都有自己的页目录表和页表，当任务创建时，它们一同被创建。当任务执行时，页部件使用它们访问任务自己的私有内存空间（页面）。但是，任务的页目录表和页表不能只包含任务的私有页面。如果不是这样，当任务调用内核服务时，或者换句话说，进入0 特权级的全局地址空间执行时，地址转换将无法进行，因为任务的页目录表和页表里没有登记内核所占用的那些物理页面。

还记得吗，我们一直在说，任务的4GB 地址空间包括两个部分：局部空间和全局空间，全局空间是所有任务共用的。很明显，内核就是所有任务共用的，它应当属于每个任务的全局空间。

一般来说，公平起见，全局地址空间占据着任务4GB 地址空间的高2GB，对应的线性地址范围是0x80000000～0xFFFFFFFF；而局部地址空间则使用低2GB，对应的线性地址范围是0x00000000～0x7FFFFFFF。如图16-15 所示，地址空间的分配必须在每个任务的页目录中体现，页目录的前半部分指向任务自己的页表；后半部分则指向内核的页表。否则的话，当转到内核中执行时，是无法完成地址转换的，因为找不到对应的目录项和页表项。

在任何任务内，在任何时候，如果段部件发出的线性地址高于等于0x80000000，指向和访问的就是全局地址空间，或者说内核。

为此，我们要修改内核自己的页目录表，甚至是内核各个段的描述符，将内核挪到虚拟地址空间的高端，也就是虚拟地址空间中，从0x80000000 开始的一段连续区域。也许你并未安装这么多物理内存，但是，没有关系，我都说了，这是线性地址空间，或者叫虚拟地址空间。

如图16-16 所示，这是映射到虚拟内存高端地址后的内核布局图。

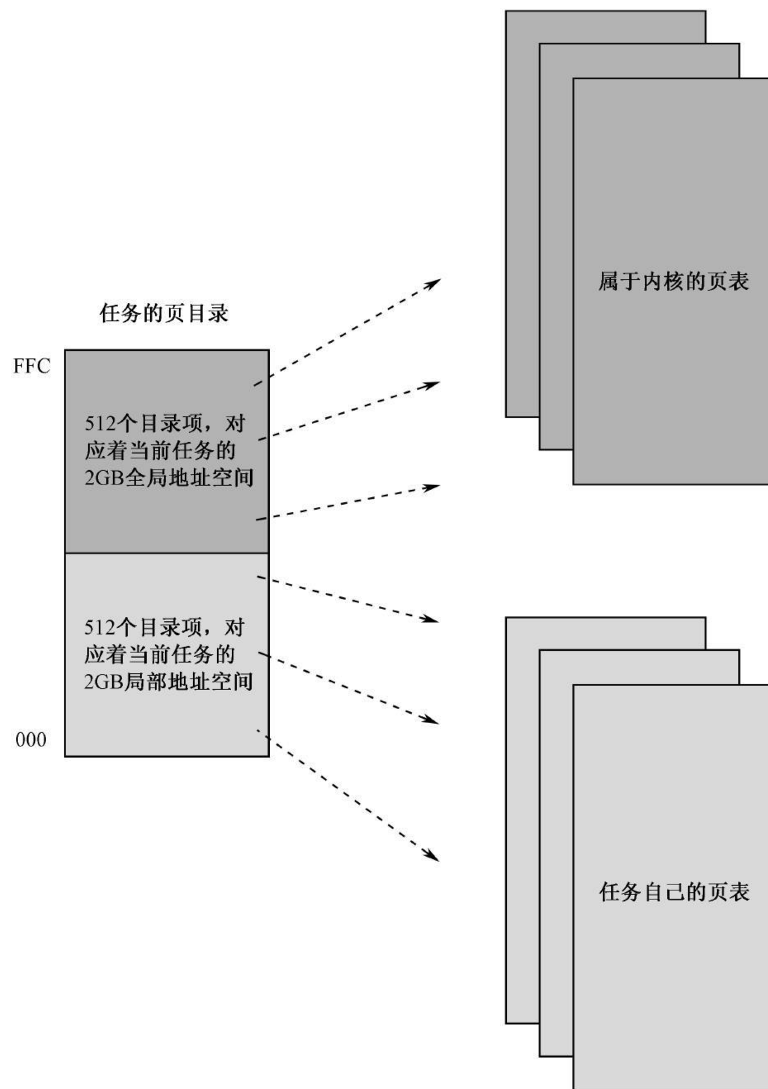


图16-15　通过页目录和页表来实现地址空间的分配

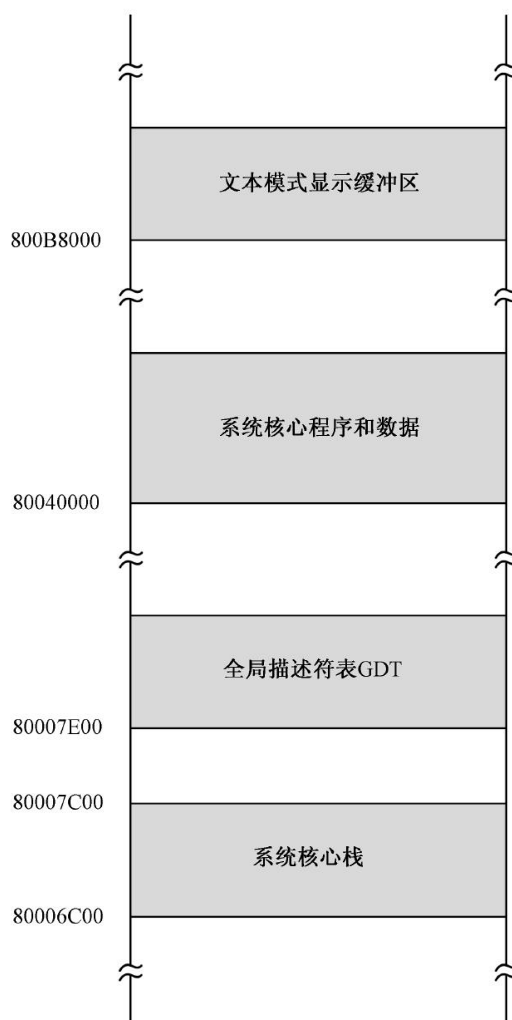


图16-16　映射到高端地址后的系统核心布局

第969～973行，在内核的页目录表中，创建一个和线性地址0x80000000对应的目录项，并使它指向同一个页表。毕竟，我们只改变了线性地址空间范围，内核的数据和代码仍然在原来的页内，没有改变。

为了修改页目录表PDT，需要访问它，知道它的物理地址。但是，当前已经开启了分页功能，在分页机制下，程序只能使用线性地址，访问内存必须先访问页目录和页表，通过它们转换之后的地址才是能够发送到内存芯片的物理地址，你自己知道页目录表的物理地址，这没有用。或者，说得更清楚一点，你访问的是页目录表，但却还要通过页目录表进行地址转换之后才能访问内存中的页目录表。这有点自相矛盾，

除非页目录表中有一个目录项能指向页目录表自己。否则，访问一个并未在页目录表和页表内登记的页，会引发处理器异常中断。

这段代码，其实倒过来，先从结果着手可能更容易理解。经过分析可知，当第973 行的指令

```
mov dword [es:ebx+esi],0x00021003
```

执行时，ES 是指向0～4GB 内存段的，EBX 寄存器的内容为0xFFFFF000，ESI 寄存器的内容为0x00000200，因此，段部件发出的线性地址是0xFFFFF200。

如图16-17 所示，当前程序或者任务的页目录表，其物理基地址是由控制寄存器CR3 指示的，仅高20 位有效，是多少并不重要，可以假定为0x?????000。段部件产生的线性地址是0xFFFFF200，其高10 位的值是0x3FF，这个值乘以4，结果为0xFFC。这个值同CR3 寄存器提供的页目录表物理地址相加，结果是0x?????FFC，它就是页目录表内最后一个目录项的物理地址。从此处取出一个双字，就是线性地址0xFFFFF200 所对应页表的物理地址。

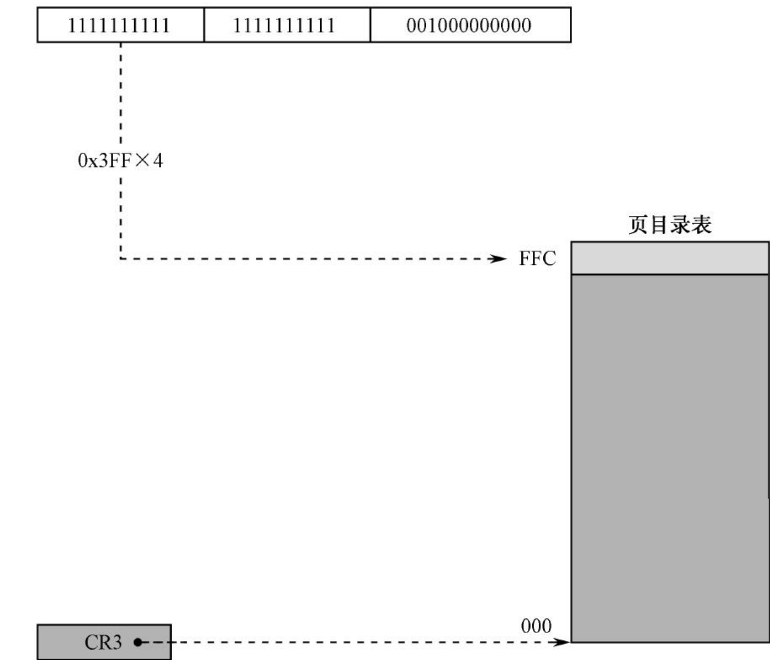


图16-17　使用线性地址访问和修改页目录表（图示一）

有趣的是，在前面第936 行，我们已经在该目录项内填写了页目录表的物理基地址。因此，该目录项所指向的页表正是当前的页目录表自

己，这实际上是把页目录表当成页表来用。

接下来，如图16-18 所示，处理器用线性地址0xFFFFF200 的中间10 位作为偏移量访问页表，这10 位的值是0x3FF。页表的物理地址就是页目录表的物理地址，即0x?????000；表内偏移量是0x3FF 乘以4，即0xFFC。故，这一次处理器发出的最终物理地址也同样是0x?????FFC，它就是页表内最后一个页表项的物理地址。从此处取出一个双字，就是线性地址0xFFFFF200 所在的那个页的物理地址。

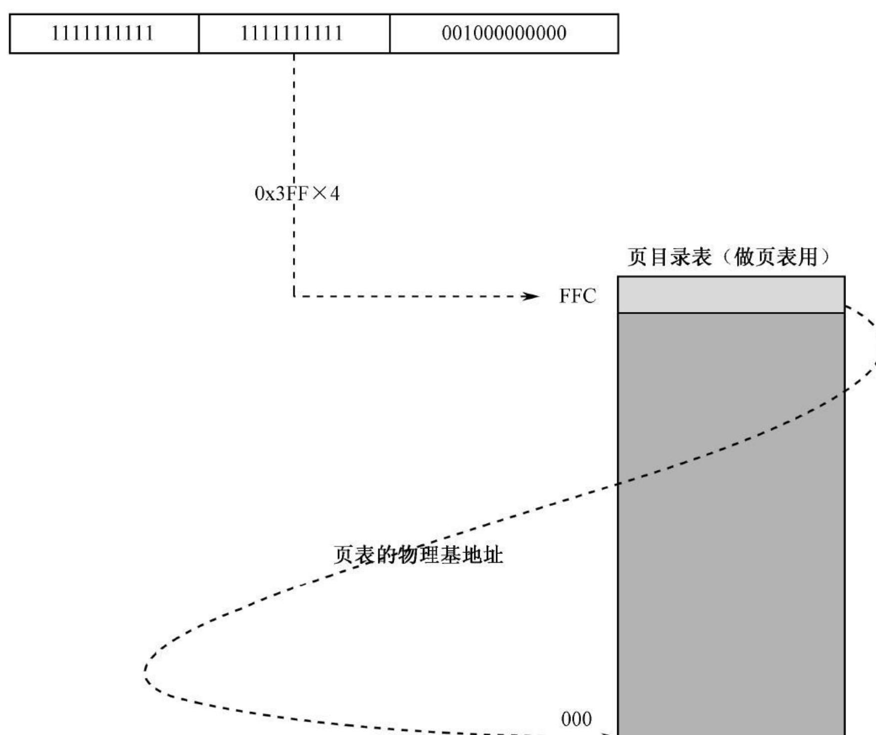


图16-18　使用线性地址访问和修改页目录表（图示二）

因为访问的又是同一内存位置，故最终要访问的页仍是页目录表自己。不过，如图16-19 所示，这一次稍有不同，页的物理地址是0x?????000，页内偏移由线性地址的低12 位乘以4 给出，其值为0x800。所以，当指令

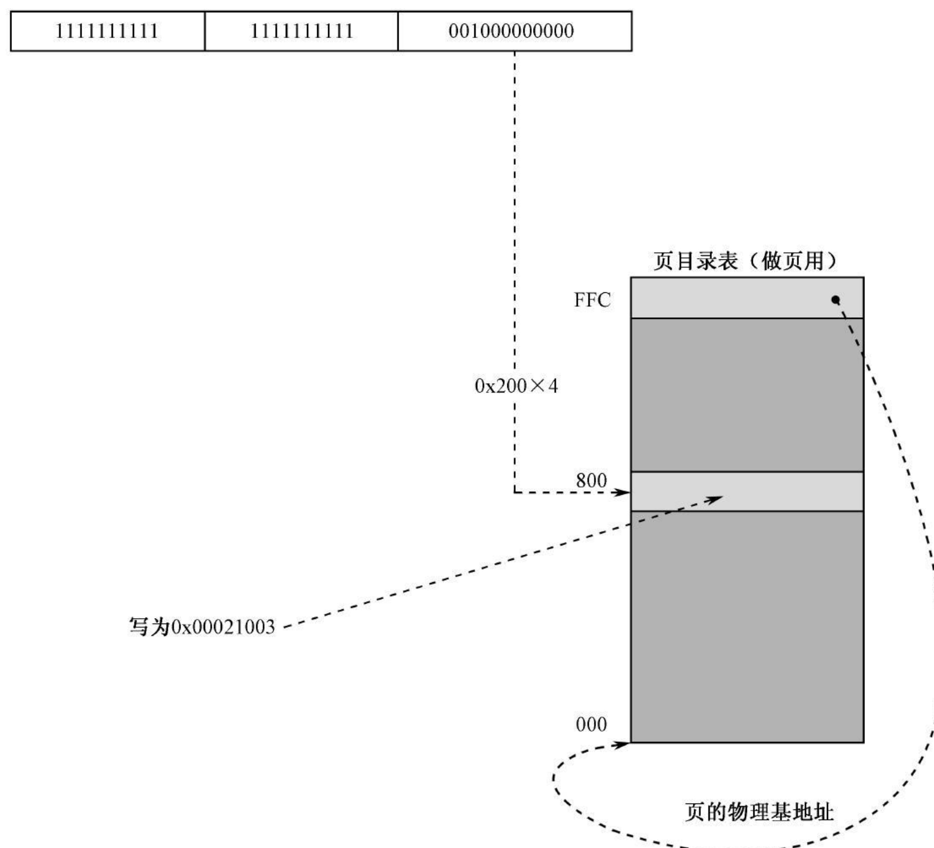


图16-19　使用线性地址访问和修改页目录表（图示三）

```
    mov dword [es:ebx+esi],0x00021003
```

执行时，处理器发出的物理地址是0x?????800，并将该双字单元的内容改写为0x00021003。

综合上面的分析，这就是说，如果页目录表的最后一个目录项指向当前页目录表自己，那么，无论任何时候，当线性地址的高20 位是0xFFFFF 时，访问的就是页目录表自己。

修改页目录表的原理就是这样。因此，当我们回过头去看时，第969 行，实际上给出了当前正在使用的页目录表的线性基地址0xFFFFF000；第970 行，给出了要修改的那个目录项所对应的线性基地址，其高10 位的值乘以4，决定了该线性地址所对应的页目录表内偏移量。因此，第971 行，将线性地址右移22 位，只保留高10 位；第972 行，再将它左移2 位，相当于乘以4。

最终，页目录表内有两个目录项都指向同一个页表，如图16-20 所示。不过，尽管指向的是同一个页表，这两个目录项所映射的线性地址

是不一样的，旧表项依然对应着线性地址0x00000000～0x000FFFFF；新表项则对应着一个高端的地址范围0x80000000～0x800FFFFF。

这回，你应该很清楚了，为什么处理器会使用层次化的分页结构，而不是用4MB 内存组建单一的页映射表。如果采用后者，将不得不至少保留2MB 的内存空间。当然，这对于现在的计算机来说算不了什么，但是，在1978 年，80386 处理器刚刚问世的时候，拥有2MB 物理内存还是一种非常奢侈的想法。即使是在15 年之后的1993 年，在我使用的计算机上也才有2MB 物理内存，已经是相当不错的，那台计算机花了7000多元人民币。

仅仅修改页目录表是没有用的，如果段部件给出的线性地址并不在0x80000000 以上，是没有用的。因此，必须修改与内核有关的段描述符，包括全局描述符表（GDT）自己的线性地址。一旦开启页功能，除页目录表和页表的地址外，其他所有地址都是线性地址，即使是在访问GDT 和LDT 的时候，内核就更不用说了，不可能因为它靠近硬件就能搞特殊。

修改段描述符很简单，只需要将其中的基地址部分加上0x80000000即可。比如GDT，原先的地址是0x00007E00，现在则要改为0x80007E00。说起来容易做起来难，段描述符中的基地址不是连续的，处于高低两个双字中的不同位置，重新计算比较麻烦。

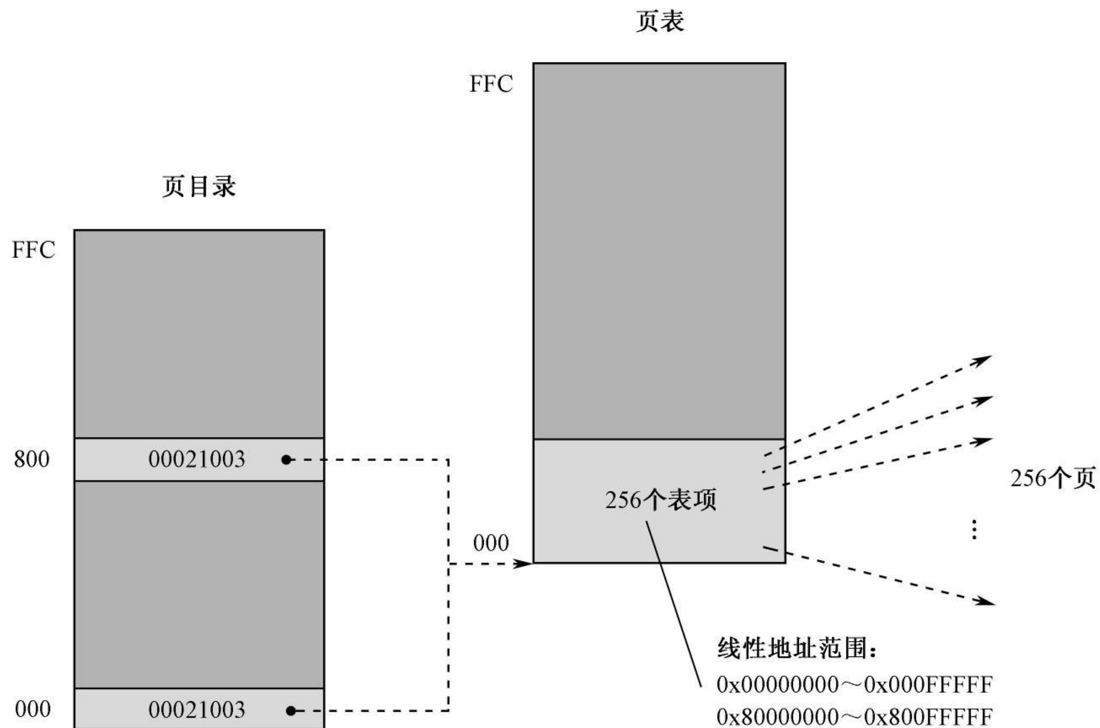


图16-20　将内核映射到高端地址后的页目录表

幸运的是，0x80000000 是一个有趣的数，仅最高位是“1”，其余31比特都是“0”。因此，只需要访问全局描述符表（GDT），将所有描述符高字部分的最高位置“1”即可。

第977～979 行，先取得GDT 的线性基地址，并传送到EBX 寄存器，准备开始访问GDT 内的段描述符。

第981～986 行，依次找到内核栈段、文本模式下的视频缓冲区段、公共例程段、内核数据段和内核代码段的描述符，并将每个描述符的最高位改成“1”。在这里，EBX 寄存器提供了GDT 的基地址；0x10、0x18、0x20 等这些数提供了每个描述符在表内的偏移量；在偏移量的基础上加4，就是每个描述符的高32 位。唯一没有修改的是0～4GB 内存段的描述符，它本身就是为访问整个内存空间而存在的，不需要修改。

尽管全局描述符表（GDT）很重要，但处理器不会对它有任何照顾，开启分页功能后，访问GDT 也同样需要使用线性地址。因此，第988 行，将GDT 的基地址映射到内存的高端，即加上0x80000000。第990 行，将修改后的GDT 基地址和界限值加载到全局描述符表寄存器（GDTR），使修改生效。

我知道，很多人会有一个疑问：在修改段描述符的时候，以及重新设定了GDT 基地址的时候，会不会导致程序的执行出现问题，毕竟访问内存需要先访问GDT，然后访问GDT 内的描述符，而你已经改变了它们。

答案是不会。在你能够访问GDT，或者能够修改描述符的时候，段寄存器CS、SS、DS、ES、FS 和GS 已经指向了相应的段，对不对？

那么，我们知道，段寄存器实际上由段选择器和描述符高速缓存器组成。当取指令和执行指令时，或者访问内存中的数据时，处理器不会每次都重新加载段寄存器，而是使用CS、SS、DS、ES、FS 和GS 描述符高速缓存器中的内容。

所以，当你改变了GDT 的基地址，或者修改了段描述符之后，这些修改不会立即反映到段寄存器的描述符高速缓存器，对程序的运行没有任何影响。

但是，当执行一个段间转移指令，或者往段寄存器里加载一个新的段描述符选择子时，处理器将会访问GDT 或者LDT，并刷新段寄存器描述符高速缓存器的内容。因此，为了使处理器转移到内存的高端位置执行，需要显式地刷新段寄存器的内容。

代码段寄存器CS 的刷新一般使用转移指令完成。因此，第992 行，使用远转移指令jmp 跳转到下一条指令的位置接着执行。这将导致处理器用新的段描述符选择子core_code_seg_sel（0x38）访问GDT，从中取出修改后的内核代码段描述符，并加载到其描述符高速缓存器中。同时，这也直接导致处理器开始从内存的高端位置取指令执行。

第995～999 行，重新加载段寄存器SS 和DS 的描述符高速缓存器，使它们的内容变成修改后的数据段描述符。注意，这些段在内存中的物理位置并没有改变。特别是栈段，因为仅仅是线性地址变了，栈在内存中的物理位置并没有发生变化，所以栈指针寄存器ESP 仍指向正确的位置。段寄存器ES 没有修改，因为它指向整个0～4GB 内存段，内核需要有访问整个内存空间的能力。段寄存器FS 和GS 没有使用。

第1001、1002 行，显示一条消息，告诉屏幕前的人，已经开启了分页功能，而且内核已经被映射到线性地址0x80000000 以上。需要特别说明的是，即使是在分页机制下，相对于上一章，过程put_string 及其嵌套过程put_char 也没有做任何修改，但依然工作得很好。原因很简单，尽管文本模式的显示缓冲区基地址已经映射到一个较高的地址

0x800B8000，但是，向该区域内的任何一个单元，比如线性地址0x800B8020 写字符时，页部件最终会在页表内找到显示缓冲区所在的那个页，页的物理地址是0x000B8000。用页的物理地址加上12 位的页内的偏移量0x020，就是最终的物理地址0x000B8020。

第1005～1023 行，安装供用户程序使用的调用门，并显示安装成功的消息。这一段代码和上一章相同，没有做任何修改。门描述符只涉及目标代码段的选择子和偏移量，但是你应该清楚，目标代码实际上已经被映射到内存的高端了。

16.4　创建内核任务

16.4.1　内核的虚拟内存分配

接下来的工作是使内核的一部分成为任务，并为创建用户任务和实施任务切换做准备。

首先是创建内核任务的任务状态段（TSS）。内核的主体部分占据着从线性地址0x80000000 开始的1MB 内存空间，即0x80000000 ～ 0x800FFFFF， 在此之后的空间， 即0x80100000 ～ 0xFFFFFFFF，是可以自由分配的。

为了连续地、动态地分配内核的空间，内核需要记住下一个可用于分配的线性地址。为此，代码清单16-1 的第529 行，专门声明了标号core_next_laddr，并初始化了一个双字0x80100000，这就是初始的可分配线性地址。每当分配了新的内存空间后，该双字将修正为下一个可分配的地址。

在分页机制下，内存的分配既要在虚拟内存空间中进行，还要在页目录表和页表中进行。原因你是清楚的，线性地址最终要通过页目录表和页表转换成物理地址，如果没有分配一个物理页，对任何内存的访问都是无效的，会引发处理器异常中断。

第1026 行，先访问内核数据段，从标号处取得当前可用的线性地址，将来要作为内核TSS 的起始线性地址。然后，将EBX 寄存器中的线性地址作为参数，调用过程alloc_inst_a_page 去申请一个物理页。

该过程位于第358 行，是公共例程段内的过程，它的功能是在可用的物理内存中搜索空闲的页，并将它安装在当前的层次化分页结构中（页目录表和页表）。简单地说，就是寻找一个可用的页，然后，根据线性地址来创建页目录项和页表项，并将页的地址填写在页表项中。

该过程首先察看与该线性地址相对应的页目录项是否存在。线性地址的高10 位是页目录表的索引，将该值乘以4，就是当前页目录表中，与该线性地址对应的页目录项。第370 行，将EBX寄存器中的线性地址传送到ESI 寄存器作为副本；第371 行，用AND 指令保留线性地址的高

10位，其他各位清零；第372 行，得到该线性地址在页目录表中对应的偏移量（目录项）。该指令等效于

```
shr esi,22        ;得到线性地址高10 位的值
shl esi,2         ;乘以4
```

无关的位已经清零，在这种情况下，右移22 次，再左移2 次，干脆右移20 次好了。

我们说过，可以用线性地址来访问当前页目录表，我们也知道，当前页目录表的线性地址是0xFFFFF000。因此，第373 行，将刚才计算出的偏移量和页目录表的线性基地址相加，得到的结果就是要访问的那个目录项的线性地址。

第375、376 行，测试该目录项的P 位，看它是否为“1”。如果为“1”，则表明对应的页表已经存在，只要在那个页表中添加一项即可；否则，必须先创建页表，并填写页目录项。

注意，尽管我们给出的就是线性地址，但是，那不是处理器段部件产生的线性地址，第366、367 行，令段寄存器DS 指向0～4GB 的内存段（段的基地址是0x00000000），此后，当我们用给出的“线性地址”作为段内偏移量访问内存，段部件才会输出真正的线性地址，尽管两者是相同的。总之，处理器的段管理机制是始终存在的，没有任何一种方法可以关闭它。

如果对应的页目录项不存在，那么，将执行第379～381 行的指令，以分配一个物理页作为页表，并将页的物理地址填写到页目录项内。为此，需要调用另一个过程allocate_a_4k_page 以得到一个可用的4KB页。

16.4.2　页面位映射串和空闲页的查找

尽管每个任务都拥有4GB 虚拟内存空间，也可以自由分配这些空间，但是，物理内存是有限的，或者用页的视角来说，物理页的数量是有限的。

写这本书的时候，拥有4GB 的物理内存并不是一件值得羡慕的事情。但是，所谓水涨船高，要知道，现在的程序也极其庞大，而且往往

都在内存中同时运行着。为了分配页，需要跟踪哪些页已经分配，哪些页是空闲的，这对操作系统来说是必做的事情。

很容易想到，操作系统必须在刚刚获得计算机控制权的时候，就检测实际的物理内存数量，并建立一张表格，标明页的物理地址及其是否空闲。当有程序申请内存时，就寻找这样的空闲页，并将其标记为已分配。

内存空间来自于插在主板上的内存条，按照新的工业标准，每个内存条上焊有一个很小的只读存储器，用于标明该内存条的容量和工作参数。作为一个PCI(E)设备，软件可以读取它，以获得计算机上的物理内存容量。然后建立上述的页分配表。

如果你的计算机上真的有4GB 物理内存，那么，它可以划分为1048576（220）个页。如果每个表项占一字节，则需要1MB 内存来创建该表。显然，这有些不划算。为了简单，可以使用位串来指示页的分配情况。

如图16-21 所示，可以用一个长的比特串，叫做页映射位串，来指示每个页的位置及分配情况。取决于你所拥有的实际内存数量（页数），该串最多可以有1048576 比特，由于每字节包含8个比特，所以，共需要131072 字节，也就是128KB。

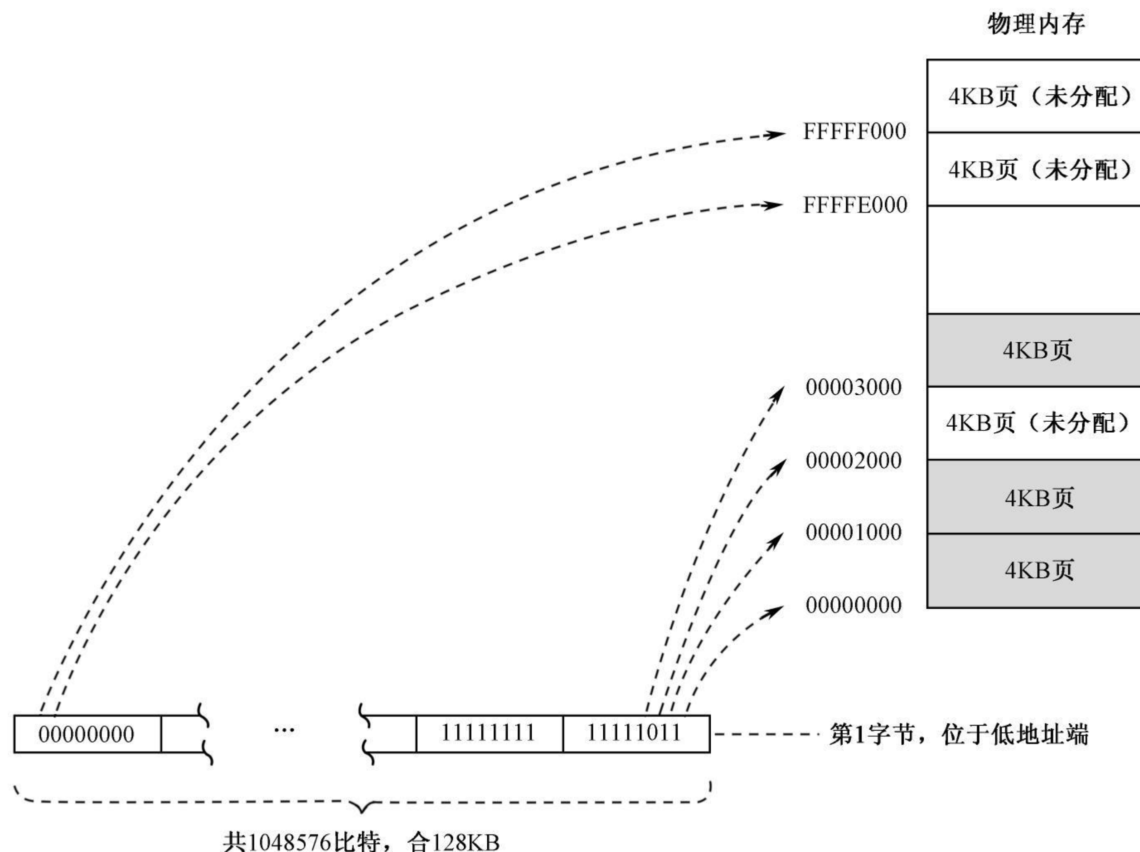


图16-21　页映射位串示意图

比特在位串中的位置，决定了它所映射的页在哪里。如图16-21 所示，位0 对应的是物理地址为0x00000000 的页，位1 对应的是物理地址为0x00001000 的页，位2 对应的是物理地址为0x00002000 的页，……，最后一个比特对应的是最后一个页，即物理地址为0xFFFFF000 的页。

除了用比特所在的位置决定页的位置外，比特的值决定了页的分配情况。当某比特为“0”时，表示它所对应的页未分配，是可以分配的空闲页；否则，就表明那个页已经被占用了，不能再分配给任何程序。

在本章中，没有检测实际可用内存的代码，仅仅假定我们只有2MB的物理内存可用。2MB的内存，可分为512 个页，需要512 个比特的位串。在实际的程序中，没有声明位串的方法，只能声明字节、字、双字等。因此，只能用连续的字节或字数据来形成位串。

回到代码清单16-1 中，第465 行，声明了标号page_bit_map，并初始化了64 字节的数据。这64 字节首尾相连，形成一个512 比特的位串。对照图16-21，第1 字节的位0 对应着物理地址为0x00000000 的页，第1

字节的位1 对应着物理地址为0x00001000 的页，……，第2 字节的位0 对应着物理地址为0x00008000 的页，以此类推。

耐心一点，仔细观察这个页映射位串，你会发现前32 字节的值差不多都是0xFF。这并不奇怪，它们对应着最低端1MB 内存的那些页（256 个页），它们已经整体上划归内核使用了，没有被内核占用的部分多数也被外围硬件占用了，比如ROM -BIOS。

当然，如果你眼很尖的话，也会发现其中混杂了两字节的0x55。这又是怎么回事呢？

回到前面，看图16-10，尽管画得不明显，但是依然能看出，在物理地址0x00030000～0x00040000 之间，是一段较为连续的空闲区，共64KB，可划分为16 个页，页的物理地址为0x00030000～0x00040000，就对应着这两字节。本来，这两字节都应当是0x00，以表明是可以分配的空闲页。不过，为了表明大的、连续的线性地址空间不必对应着连续的页，我们有意将空闲的页在物理上分开，因为0x55 的二进制形式是01010101。同样的做法也出现在后面的64 个页中。

当然，这么做未必合法，因为低端1MB 内存已经完整地分配给了内核，在内核的页表中，已经有页表项指向这16 个页。如果我们再把它分配给其他任务，那么，该任务的页表项也势必指向这16 个页，等于重复分配。不过，请放心，这16 个页内核是不会用到的，因此，分配给其他任务也无妨。

回到代码清单16-1 中，来看过程allocate_a_4k_page 是怎么搜索页映射位串并分配页的。

页映射位串位于内核数据段中。第332、333 行，先令段寄存器DS 指向内核数据段。

接着，第335～341 行，从头开始搜索位串，查找空闲的页。具体地说，就是找到第一个为“0”的比特，并记下它在整个位串中的位置。搜索位串用到了指令bts。

bts（Bit Test and Set）指令测试位串中的某比特，用该比特的值设置EFLAGS 寄存器的CF 标志，然后将该比特置“1”。它最基本的两种格式为

```
bts r/m16,r16
bts r/m32,r32
```

在这里，目的操作数可以是16/32 位的通用寄存器，或者指向一个包含了16/32 位实际操作数的内存单元，用于指定位串；源操作数可以是16/32 位的通用寄存器，用于指定待测试的比特在位串中的索引（位置）。

如果目的操作数是通用寄存器，那么，指定的位串就是该寄存器的内容（长度为16 比特或者32 比特）。在这种情况下，根据操作数的长度，处理器先求得源操作数除以16 或者32 的余数，并把它作为要测试的比特的索引。然后，从位串中取出该比特，传送到EFLAGS 寄存器的CF 位。最后，将该比特置位。

则如果目的操作数是一个内存地址，那么，它给出的是位串在内存中的起始地址，或者说该位串第1 个字或者双字的地址。同样地，源操作数用于指定待测试的比特在串中的位置。因为串在内存中，所以其长度可以最大程度地延伸，具体的长度取决于源操作数的尺寸，毕竟它用于指定测试的位置。如果源操作数是16 位通用寄存器，位串最长可以达到216 比特；如果源操作数是32 位的通用寄存器，则位串最长可以达到232 比特。无论如何，在这种情况下，指令执行时，处理器会用目的操作数和源操作数得到被测比特所在的那个内存单元的线性地址。然后，取出该比特，传送到EFLAGS 寄存器的CF 位。最后，将原处的该比特置位。

除此之外，这两种指令格式的区别还在于具体操作时，处理器读取的数据的长度。挑选比特的工作是在处理器内部进行的，要先从内存中读取含有指定比特的字或双字。第一种指令格式进行的是16 位的内存操作，处理器读的是一个字；第二种指令格式进行的是32 位的内存操作，处理器读的是一个双字。

bts 指令并不孤独，同类型的指令还有btr、btc 和bt 它们的区别如表16-1 所示。

表16-1　bts/btr/btc/bt 指令对照表

指　　令	英 文 全 称	基 本 功 能	对其他标志位的影响
bts	Bit Test and Set	将指定位置的比特传送到 CF 标志位，然后将其置位	ZF 标志位不受影响，对 OF、SF、AF 和 PF 标志的影响未定义
btr	Bit Test and Reset	将指定位置的比特传送到 CF 标志位，然后将其复位（清零）	
btc	Bit Test and Complement	将指定位置的比特传送到 CF 标志位，然后将其取反	
bt	Bit Test	将指定位置的比特传送到 CF 标志位	

回到代码清单16-1。

搜索空闲页是一个机械的工作，要先从位串的第1 个比特开始。第335 行，先将EAX 寄存器清零，这表明我们要从位串的第1 个比特开始搜索。

第337 行，执行bts 指令。这将使指定的比特被传送到标志寄存器的CF 位，同时那一位被置“1”。置“1”是必做的工作，如果它原本就是“1”，这也没什么影响；如果它原本是“0”，那么，它就是我们要找的比特，它对应的页将被分配，而将它置“1”是应该的。

第338 行，判断位串中指定的位是否原本为“0”。如果答案是肯定的，那么，太好了，于是转到第348 行执行，准备退出当前过程；如果不是，那么，第339～341 行，将EAX 的内容加一，准备测试位串中的下一比特。在此之前，要先判断是否已经测试了位串中的所有比特，以防止越界。page_map_len 是一个用伪指令equ 声明的常数，位于第473 行，它的值就是位串的字节数。将它乘以8，就是位串的比特数。在最坏的情况下，没有找到可以用于分配的空闲页，则显示一条错误消息，并停机。当然，对于一个流行的操作系统来说，这样做是不对的，正确的做法是看哪些已分配的页较少使用，然后将它换出到磁盘，腾出空间给当前需要的程序，到时候再换回来。不过，这已经超出了本书的主题范围。

情况乐观时，会找到一个可以分配的空闲页，也就是一个为“0”的比特。第348 行，将该比特在位串中的位置数值乘以页的大小0x1000（或者十进制数4096），就是该比特所对应的那个页的物理地址。

找到了可用的页，任务也就完成了。第355 行用于返回到当前过程的调用者。可以看出，这是公共例程段内的内部过程，仅供同一段内的其他过程使用。返回时，页的物理地址位于EAX 寄存器中。

16.4.3　创建页表并登记分配的页

返回点位于过程alloc_inst_a_page内，本次调用allocate_a_4k_page过程的目的是分配一个页作为页表。页表的地址要登记在页目录表内，仅高20位有效，对应着页表物理地址的高20位，页表地址的低12位是页表属性。从第380行可以看出，页的属性值是0x007，即，US＝1，特权级别为3的程序也可以访问；RW＝1，页是可读可写的；P＝1，页已经位于内存中，可以使用。内核的页表为什么允许特权级别为3的程序访问呢？当然了，原则上是不允许的，但是，这个例程既要用于为内核分配页面，也要用于为用户任务分配页面。对于前者，要求将所分配页面的U/S位清“0”；对于后者，要求将所分配页面的U/S位置“1”，这两者难以兼顾。为了不把事情搞复杂而又能说明问题，用当前过程所分配的页面，US位一概设置成“1”。

刚分配的页是作为页表使用的，它应当登记在页目录表内，作为目录项存在。现在，ESI寄存器中的内容就是该目录项的线性地址。第381行，将目录项的内容修改为页表的物理地址。

过程alloc_inst_a_page的功能是根据给定的线性地址，设置页目录表和页表的内容。因此，只有当页表不存在的时候，才动态分配一个页表。无论如何，现在页表已经有了，剩下的工作就是为那个线性地址分配一个最终的页，并登记在页表内。为此，需要访问页表。这同样是一个两难的问题，在分页机制下，访问内存需要通过页目录表和页表，而我们访问的正是页表。

这可如何是好？

不管页表是原来就有，还是刚才创建的，程序的执行流程最终会到达第385行。因为用于分配页的线性地址位于EBX寄存器中，这一行用于在ESI寄存器中制作它的一个副本。

因为是要修改页表内的页表项，所以，无论如何，必须要知道该页表项的线性地址才行，这可以分几步来完成：

首先，我们知道，ESI寄存器中的线性地址，其高10位决定了页表在页目录表中的登记位置；中间10位，决定了页在页表中的登记位置。很显然，要访问页表，就得把页表当成普通页来访问。如此一来，那个页表项在页表中的位置，就相当于数据在页中的位置。为此，应当把ESI

寄存器的中间10 位乘以4 之后，挪到该寄存器的低12 位，作为页内偏移量；

其次，既然把页表作为普通的页来对待，那么，页部件势必要先访问该“页”的“页表”。页表的物理地址是登记在页目录表中的，它在页目录表中的位置由ESI 寄存器的高10 位指定，因此，就得把页目录表当成该“页”的“页表”来用，并把ESI 寄存器的高10 位挪到中间10 位上，作为页表项的索引号。

最后，为了将页目录表作为页表来用，要将ESI 寄存器的高10 位置成0x3FF。这意味着，页目录表内最后一个目录项就是页表的物理地址。又因为该目录项又指向页目录表自身，故，等于是又把页目录表当成页表来用。至此，任务完成，ESI 寄存器中得到的新值，就是要修改的那个页表项的线性地址。下面结合代码清单来讲一讲具体的做法。

第386～388 行，将ESI 寄存器中的内容右移10 次，清除两边，只保留中间的10 位，同时，将高10 位的内容改成二进制的1111111111（0x3FF）。这样一来，当页部件进行地址转换时，它用高10 位的0x3FF 乘以4 去访问页目录表。由于此表项存放的是页目录表自己的物理地址，因此，此表项所指向的页表，正是当前页目录表自己，这实际上是把页目录表当成页表来用。

接着，页部件又用中间的10 位去访问页表，其实就是访问页目录表自己。于是，它就得到了页的物理地址，其实就是页表的物理地址。很好，现在只需要低12 位的偏移量就可以了。因为是把页表当成普通的页来访问，因此，需要把原线性地址的中间10 位当成页内偏移量来用。

这就是说，现在ESI 寄存器中的内容，就是页表的线性地址。

调用者传入的线性地址依然完好地保存在EBX 寄存器中，到了过程的最后，也不必制造它的副本了，直接用吧。第391 行，用and 指令只保留中间的10 位，两边清零；第392 行，将它右移12 次，再乘以4（左移2 次），作为表内偏移量。因为无关位都已清零，故可以直接写成

```
shr ebx,10
```

页表的线性地址位于ESI 寄存器中，现在，第393 行，将它和EBX 寄存器中的偏移量相加（合并），就是要修改的那个页表项的线性地址。

要修改的位置找到了，但页还没有分配呢。第394 行，调用过程allocate_a_4k_page 分配一个页，并在EAX 寄存器中返回页的物理地址。第395 行，为其添加属性值0x007。第396 行，将页表项的内容修改为页的物理地址。

至此，给出一个起始的线性地址，分配一个页，并登记在层次化的分页结构中，任务完成。第403 行，retf 指令将控制返回到调用者。

16.4.4 创建内核任务的TSS

从过程allocate_a_4k_page 返回后，返回点位于代码清单16-1 的第1028 行。

在为系统内核的任务状态段（TSS）分配了虚拟地址空间和页之后，这一行将标号core_ next_laddr 处的数据修改为下一个可分配的起始线性地址。下一次在内核的虚拟地址空间里分配内存时，将使用这个新值作为起始的线性地址。

第1031～1038 行，填写和初始化TSS 中的静态部分，有些内容，比如CR3 寄存器域，对任务的执行来说很关键，必须事先予以填写。

一旦分配了物理页，并填写了页目录项和页表项，就立即可以用那个线性地址来访问内存。此时，整个过程是相反的，页部件用那个线性地址访问页目录表和页表，生成物理地址。还有，尽管你在指令中给出的确实是线性地址，但并非是由段部件生成的线性地址。在Intel 处理器上，段机制是无法关闭的，因此，你必须使用0～4GB 的段，加上你的“线性地址”，才得使段部件生成真正的线性地址，尽管两个线性地址在数值上没有任何不同。

因此，要访问TSS，必须通过段寄存器ES 所指向的0～4GB 数据段。

第1041～1046 行，创建内核任务的TSS 描述符，并安装到GDT中。TSS 描述符选择子保存在内核数据段中，位于第530 行，在那里，声明了标号program_man_tss 并初始化了1 个字。在任务切换时，需要使用它。

由于当前任务事实上正处于运行中，因此，只要后补手续即可使其完全合法。第1050 行，将当前任务的TSS 描述符传送到任务寄存器

TR。

16.5　用户任务的创建和切换

16.5.1　多段模型和段页式内存管理

一直以来，我们都工作在分段的内存管理模型上。如图16-22 所示，在保护模式下，首先按程序的结构分段，创建各个段的描述符，用描述符指向物理内存中的各个段。描述符中的基地址给出了段的起始物理地址，界限值给出了段的长度（边界），属性值指示了段的类型和特权级别等性质。

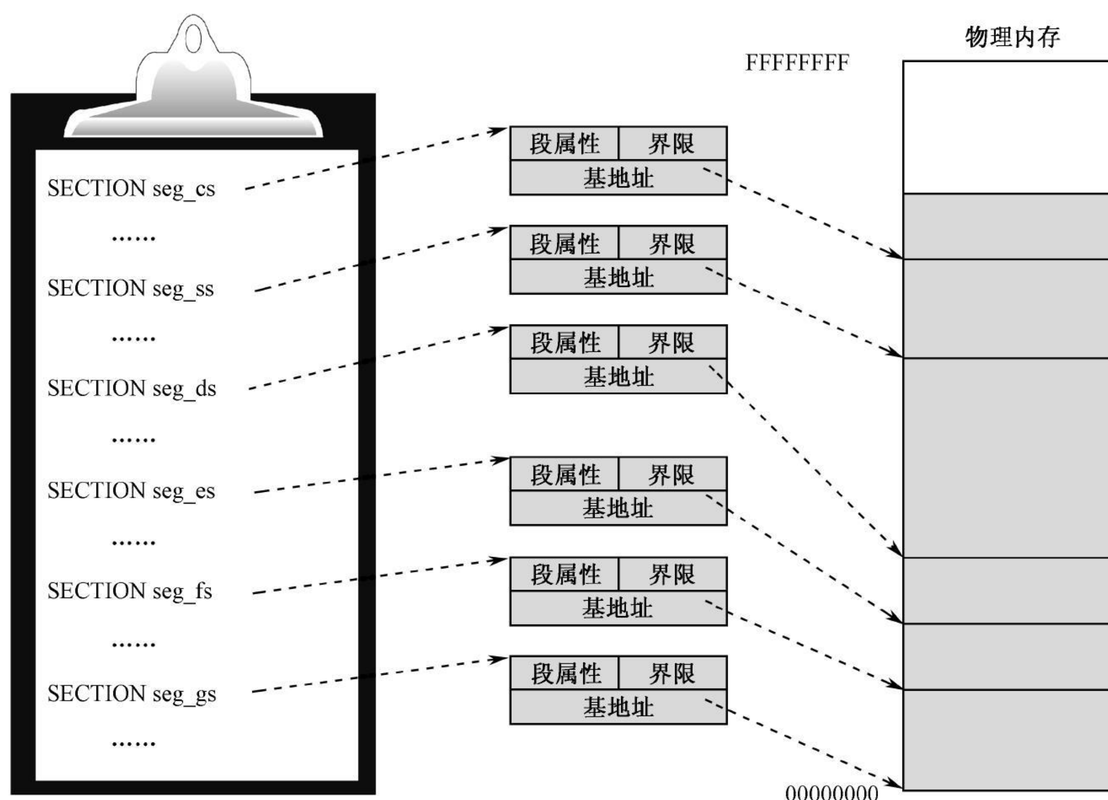


图16-22　传统的多段模型示意图（未开启页功能）

传统的多段模型（Multi-Segment Model）适用于开启了页功能之后的系统环境。如图16-23 所示，首先依然是按程序的结构分段，创建各个段的描述符。但是，段是在任务自己的虚拟地址空间内分配的，而不是在物理内存中分配的。因此，段描述符中的基地址是段的线性地址，或者说是虚拟地址。

因为开启了页功能，虚拟地址空间上的段要映射到物理内存中的一个或多个页。段是连续的，但它所占用的页不要求是相邻的。在未开启页功能之前，段基地址和段偏移相加产生的线性地址就是物理地址，开启页功能之后，线性地址还要经页部件转换后，才能得到实际的物理地址。

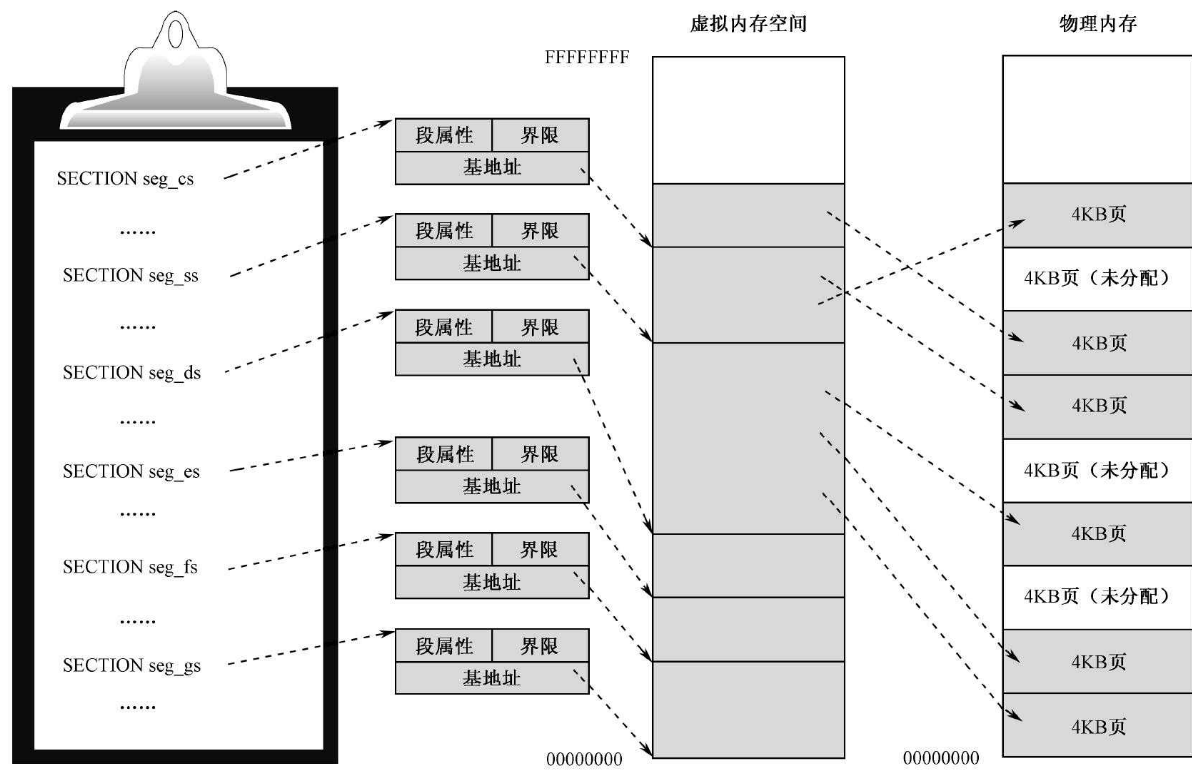


图16-23　分页机制下的多段模型示意图

为什么要分段？这是个问题。

分段的做法是随着8086 处理器的流行和广泛应用而兴起的。那个时候，处理器是16 位的，只能处理16 位的地址数据，因此，可访问的内存空间是64KB。为了访问1MB 的内存，只能分段。如此一来，可以将控制从一个段转移到另一个段，也可以通过将新段的基地址加载到数据段寄存器（DS 和ES），来访问另一个段中的数据。总之，通过这种笨拙的、变通的、迂回的方法，就能间接地获得访问1MB 内存的能力。

在8086 处理器上增加分段机制还有一个额外的好处，那就是可以用很简单的方法实现程序的重定位，让程序在内存中的位置自由浮动，而又不影响它的访问和执行。但是，这只是一个附加的礼物，因为即使不分段，也有办法实现程序的自由浮动和重定位，只是肯定会麻烦很多。

任何事情只要一流行，就会被认为是必然的，而不管它事实上有多不合理。到了32 位处理器时代，分段的方法依然被完整地保留下来了。也许是真的动了脑筋、花了心思，处理器的设计者居然找到了分段模型的好处，那就是可以防止一个程序访问不属于自己的段。

但是，由于分页功能的出现，弱化了人们关于分段机制是否合理的信心。内存的访问是通过页目录表和页表进行的，每个任务都有自己的页目录表和页表，操作系统控制着物理页的分配权，除非它把一个页分配给某个任务，并填写到那个任务的页目录表和页表里，否则，那个任务不可能拥有访问那个内存位置的能力。

尽管处理器的设计者一直在宣称，把分段和分页机制结合在一起，将获得最大强度的保护功能，但是，事实上，在现实的软件设计者那里，多段模型已经不那么吃香了。

典型地，32 位的处理器拥有32 根地址线和32/64 根数据线，这使得它不用将4GB 或多于4GB 的内存空间划分成多个段，就能完全控制它。如此一来，软件设计者就会倾向于不分段。当然，程序的浮动和重定位将不可能再依赖于分段机制，但并不是没有其他办法。

16.5.2　平坦模型和用户程序的结构

不分段的内存管理模型称为平坦模型（Flat Model）。尽管说是不分段，但你千万不要信以为真，分段是Intel 处理器的固有机制，处理器总是按“段地址＋偏移量”来形成线性地址，不可能绕开这种工作机制。

因此，如图16-24 所示，所谓的平坦模型，就是将全部4GB 内存整体上作为一个大段来处理，而不是分成小的区块。在这种模型下，所有段都是4GB，每个段的描述符都指向4GB 的段，段的基地址都是0x00000000，段界限都是0xFFFFF，粒度为4KB。

在这种基本的平坦模式下，程序在编写的时候不分段，即，只保留一个段，代码和数据都在这个段内，相互邻接，但一般并不交叉。很显然，在这种模式下，不能享受到段保护机制的好处，段界限和数据访问的检查仍然进行，但从不会产生违例的情况。原因很简单，每个段描述符的基地址都是0，实际使用的段界限都是0xFFFFFFFF，就任务内的地址空间而言，对任何内存位置的访问都是合法的。

一个使用基本平坦模式的实例是代码清单16-2，程序没有分段，或者说只有一个大的段。在程序中，指令和数据的偏移量只和它们出现的自然位置有关。

在这里，一个基本的特点是，所有内容都是按类型组织的。比如，一开始是和整个程序有关的统计数据，比如程序的大小、入口点、U-SALT 的大小和起始位置等，在往常，这就是用户程序头部段；然后，是程序中用到的数据，包括U-SALT。传统上，这就是数据段；最后，是可执行代码部分。

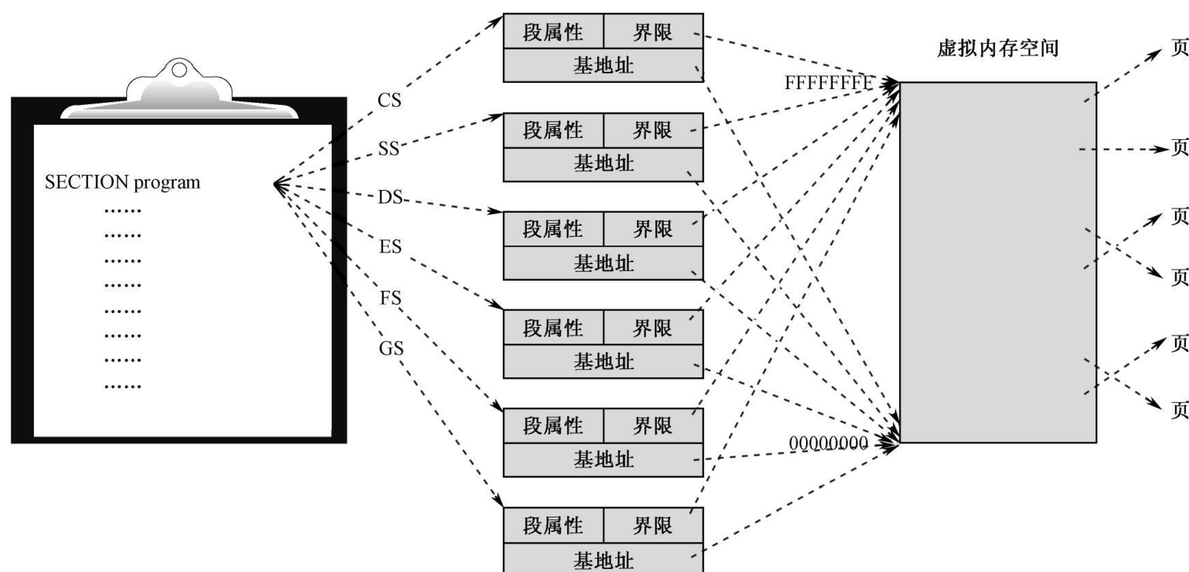


图16-24　基本的平坦模型示意图

传统上，这几个部分是以段为自然分界的。但是现在，它们混合在一起，按类型划分，而不是段，类似于几种流行的可执行文件中的节。

程序本身不大，但编译之后却很大。这是因为，第23 行，保留了一个很大的空白区域。在那里，声明了标号reserved，并初始化了128000字节。空白区的位置选择也很特别，位于U-SALT的中间，把U-SALT 表分为遥遥相望的两部分，就像牛郎和织女。这是有意的，不管是写程序，还是处理器执行程序，内存的访问都应该是线性的、连续的、连贯的，为程序分配内存时，每个页在物理上本就不相邻，现在，U-SALT 这么庞大，必定会被分散于各处。这么做，只是为了验证处理器和当前程序是否能在分页机制下正常工作。

需要注意的是，U-SALT 的大小是256 字节对齐的，因为每个表项占256 字节。因此，空白数据的大小也应当是256 的整倍数。否则，在程

序的重定位阶段，内核将不能正确地处理USALT 表。

16.5.3　用户任务的虚拟地址空间分配

现在，回到代码清单16-1，开始加载用户程序并创建相应的任务。

首先要创建任务控制块（TCB）。这本来不是个问题，但由于现在每个任务都拥有了自己独立的4GB 虚拟地址空间，问题就来了。

一般来说，所有任务的TCB 都应当占用内核的地址空间，在内核的虚拟地址空间里分配。任务都是由内核负责管理和调度的，如果TCB 位于任务自己的地址空间里，而不是内核的地址空间里，那么，这同时也意味着，在内核的页目录表和页表中，没有指向TCB 所在页的表项，内核不可能访问到它。

第1055～1057 行，用于在内核的虚拟地址空间里分配4KB 的内存（页），这和上一次在内核中分配内存的做法是一样的。

第1059～1062 行，初始化TCB，为某些域赋初值。任务控制块（TCB）的结构如图16-25 所示，和上一章相比，已经大大简化了，只保留了少数项目。这也意味着，和以往相比，用户程序的加载过程会简单得多。

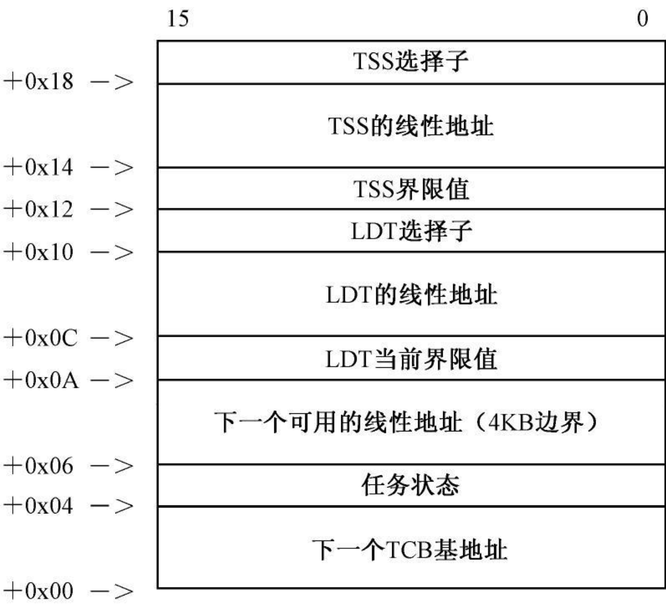


图16-25　任务控制块（TCB）的结构

在TCB 中，有两个项目应该在创建用户任务前就予以填写和初始化，它们是LDT 当前界限值和下一个可用的线性地址。LDT 当前界限值应该被初始化为0xFFFF，这是计算机启动时，LDTR 寄存器中的默认界限值，LDTR 中的界限部分只有16 位。LDT 的界限是LDT 的长度减一，LDT 的初始长度为0，因此，其界限值是0xFFFF。

每个任务都有自己的4GB 虚拟内存空间，线性地址范围是0x00000000～0xFFFFFFFF，它是任务自己的空间，可以任意分配和使用。当然，实际可以使用的空间是前2GB，后2GB 被任务的全局部分占用， 映射并指向内核的页表。一般来说， 第一个可以分配的线性地址是0x00000000，要把这个数值填写到TCB 的“下一个可用线性地址”域中。

在填写了以上两个TCB 域后，第1062 行，将此TCB 挂到TCB 链上。该链的作用将在第17章中进行抢占式任务切换时才能体现出来。

第1064～1067 行，在当前栈中压入两个参数，分别是用户程序的TCB 基地址和起始逻辑扇区号，然后调用过程load_relocate_program 加载和重定位用户程序，并创建为一个任务。

16.5.4　用户程序的加载

要加载用户程序，并将其创建为一个任务，首先要做的事就是分配内存空间。

在分页机制下，每个任务都有自己的4GB 虚拟地址空间，内存分配是在任务自己的地址空间上进行的。当然，内核可以做这些事，它为用户任务创建页目录表和页表，然后看看用户程序有多大，需要多少内存空间，根据需要分配若干物理页，并将它们登记在页目录表和页表中，以便能够访问它们。最后，将用户程序的内容从硬盘读出后写进这些页中。

可是，我们忘了一件事。

在上面，内核可以为用户任务创建页目录表和页表，也能够根据用户程序的大小分配物理页，并修改页目录表和页表，这都没问题。但是，它修改的应当是用户任务的页目录表和页表，而不是它自己的。要知道，现在是在内核中执行，处理器使用的是内核的页目录表和页表，这是由控制寄存器CR3 所决定的。在内存的访问上，处理器是公平的，

因为即使是操作系统，也不能访问未在当前页目录表和页表中登记的内存空间。

好吧，那我们可以让内核先创建用户任务的页目录表，然后，临时改变控制寄存器CR3 的内容，使其指向这个新的页目录表，并在这个新表上做那些事情。等用户程序加载完毕，新任务也创建成功，再切换到内核自己的页目录表。

然而这也会出问题。内核也要靠自己的页目录表和页表才能正常工作，一旦改变了页目录表，新页目录表还是空的，当内核访问自己的地址空间时，处理器的页部件突然发现自己熟悉的页目录项全不见了。于是，又一个处理器异常不可避免地发生了。

一个可行的方案是，创建用户任务的页目录表，并将内核页目录表中的内容复制过去。然后，切换到用户任务的页目录表上去工作。这样做是合理的，我们一直在说，每个任务都拥有4GB 的虚拟内存空间，但前2GB 是它私有的，后2GB 是全局部分，映射到内核的地址空间。也只有这样，才能使内核能够继续正常运行，同时还能在页目录表的前半部分创建任务自己私有的分页系统。

上面的方法已经足够好了，如果还要说些什么的话，那就是，还有更好的办法。即，依然在内核的地址空间上工作，或者说，依然使用内核自己的页目录表，但只修改它的前半部分，因为那里属于任务的局部地址空间。最后，再把内核的页目录表复制一份，作为用户任务的页目录表。

要做什么，现在已经清楚了。现在，回到代码清单16-1，来看看具体的实施步骤。

第585～590 行和上一章相同，做一些寄存器内容的保护工作，并为访问栈中的参数做准备。

第592～602 行，用于将当前页目录表的前半部分清空。在每次创建一个新任务时，都应当清空内核页目录表的前512 个目录项。当前页目录表的后半部分是由内核使用的，内核的虚拟地址空间被映射在每个任务的高地址端，即0x80000000 之后。我们已经知道，当前页目录表的线性地址是0xFFFFF000，这是它的起始地址，位于EBX 寄存器中。ESI 寄存器用于提供每个表项的索引号，将索引号乘以4，再和基地址相加，就能得到每个目录项的线性地址，一共要处理512 个表项。

接下来要计算用户程序的大小，为完全加载它做准备。

第605、606 行，使段寄存器DS 指向内核数据段，准备从硬盘上把用户程序的第一个扇区读入内核缓冲区。

第608～610 行，从栈中取得用户程序所在的逻辑扇区号，和内核缓冲区的首地址一起，作为参数调用过程read_hard_disk_0，读取用户程序的第一个扇区。

用户程序的第一个双字就是它的大小，这是约定好的，故，第613 行，直接将内核缓冲区的第一个双字传送到EAX 寄存器。

和以往不同，现在的内存分配是按页进行的，所以最好使程序的大小能被4096 整除。第615 行，强制使程序的大小为4096 的整倍数，如果一个数能被4096 整除，那么它的最低12 位必然全是零。如此处理之后，新数值可能比原数值小。因此，第616 行，为它增加4096 字节，多总比少好。

第617、618 行，看一下处理之前的程序大小，如果人家本来就是4096 的整倍数（低12 位全是零），那就不用新值，还用旧值；如果原先的低12 位不全是零，说明我们处理得对，应该用新值。

第620、621 行，将程序的大小右移12 次，相当于除以4096，这得到的是它占用的页数。为什么要先传送到ECX 寄存器呢？原因是，下面要用ECX 寄存器的内容来控制循环次数。

循环的目的是分配物理页，并以4KB 为单位读取用户程序来填充页。这里不是一个循环，而是两个，而且是嵌套的，即外循环和内循环。外循环负责分配4KB 页，框架如下：

```
.b2:
    mov ebx,[es:esi+0x06]            ;取得可用的线性地址
    add dword [es:esi+0x06],0x1000
    call sys_routine_seg_sel:alloc_inst_a_page
    push ecx

    ;这里是内循环的代码

    pop ecx
    loop .b2
```

在循环开始之前，已经在第627 行把ESI 寄存器的内容置为用户程序TCB 的基地址。所以，外循环先访问用户任务的TCB，在它的虚拟内存空间上分配4KB 内存，并返回该段内存的线性地址。接着，用该线性地址分配一个4KB 的页。

以上就是外循环的功能，很简单。内循环嵌套在外循环中，也要用到ECX 寄存器，所以，在开始内循环之前，先将ECX 的内容压栈保存，内循环结束之后，立即将它出栈恢复，并开始下一次外循环。

内循环的代码如下：

```
        mov ecx,8
    .b3:
        call sys_routine_seg_sel:read_hard_disk_0
        inc eax
        loop .b3
```

外循环每执行一次，内循环要完整地执行8 次。故，一开始就将ECX 寄存器的内容设为8。然后，反复调用过程read_hard_disk_0 从硬盘上读取用户程序。EAX 寄存器的内容是在第626 行设置的，是用户程序的起始逻辑扇区号，每读一个扇区后，inc 指令将其加一，指向下一个要读取的逻辑扇区。

过程read_hard_disk_0 需要两个参数，除了EAX 寄存器中的逻辑扇区号外，段寄存器DS 必须指向缓冲区所在的段，EBX 寄存器必须指向缓冲区的线性地址。EBX 寄存器中的线性地址在外循环中依靠内存分配得到，而段寄存器DS 是在第623、624 行设置的，在那里，令它指向0～4GB 的数据段。这就是为什么在ES 已经指向0～4GB 数据段的情况下，不用ES，而非得用DS的原因。

这段代码虽然简单，但有很多可说的地方。首先，分页机制下，内存是先登记，后使用的。内存在用户任务自己的虚拟地址空间上分配，过程alloc_inst_a_page 分配一个空闲的页，并登记到当前页目录表和页表中。只有这样做了之后，才能访问这段内存，才能把用户程序写进去。

其次，程序在编写和编译之后，都是连续的，在加载后不能保证这一点。页是随机分配的，在一个真实的系统中，两个页相邻的几率很小。但是，这不会影响到程序的正确执行。尽管页不是连续的，但线性

地址必须是连续的，这就够了。处理器访问数据、取指令，用的是线性地址，只要线性地址从头至尾是连续的，页部件自会生成正确的物理地址。

最后，程序的加载必须从线性地址0x00000000 开始，也就是说，必须从整个虚拟地址空间的起始处开始加载。这是不合理的，但在本书中，只能这么做。原因是，首先，处理器始终要按段地址加偏移量的方法访问内存，这是不变的，在多段模型下，段内元素的偏移量都是相对于段的开始处。在程序加载后，段描述符中的基地址，就是段实际加载的位置。也正是因为如此，多段模型下，不管段加载到哪里，都不会影响到段内元素的访问，这就是多段模型下程序可以浮动和重定位的根本原因。

相反地，在平坦模型下，名义上是不分段，但实际上是只分一个大段。在这种情况下，不管程序实际上加载到哪里，代码段、数据段和栈段，其描述符的基地址都固定为0x00000000。这样一来，利用段机制实现程序的浮动和重定位就不可能了。

举个例子，在我们的代码清单16-2 中，程序的入口点位于程序内偏移量为0x04 的地方。要取得它的数值，需要使用指令

```
mov ebx,[0x04]
```

在多段模型下，数据段描述符中的基地址，就是数据段在内存中的起始地址。如果程序加载后，数据段的基地址是0x00200000，那么，这条指令执行时，处理器发出的地址就是0x00200004，这是没有问题的。

然而，在平坦模型下，代码段、数据段和栈段描述符的基地址固定为0x00000000，而不管程序实际上加载到哪里。因此，同样的指令执行时，处理器发出的地址是0x00000004，而不是正确的地址0x00200004。

在流行的操作系统上工作，编写的程序必须符合一定的规范才能实现浮动和重定位。比如，为了在平坦模型下实现数据和代码的重定位，很多系统要求用户程序提供一个标准的重定位表，列出所有需要动态重定位的元素。程序加载后，操作系统会找到此表，用实际的加载位置修正每一个表项。

这是非常复杂的，而且显然已经超出了本书的主题范围。因此，我们要求，程序必须加载到任务虚拟地址空间的起始处。

从多段模型转移到平坦模型上工作，一开始会不太适应。这没有什么好报怨的，等你有了一定的编程经验之后，会发现多段模型简直是非常糟糕的，要在多个段之间换来换去，一会儿就迷糊了。相比之下，平坦模型非常简单，编程思路非常清晰，用起来非常舒服。当然，代价就是需要额外的重定位处理。不过，那是操作系统和编译器的工作，而不是软件开发人员的。

16.5.5　段描述符的创建（平坦模型）

第644～652 行，在内核的地址空间内分配内存，创建用户任务的TSS。任务是由内核管理的，为了能够访问得到它，必须将其创建在内核的虚拟地址空间里。为了后面的代码访问TSS，TSS 的线性基地址要登记到任务控制块（TCB）中。

第655～658 行，创建用户任务的局部描述符表（LDT）。LDT 是任务私有的，要在它自己的虚拟地址空间里分配所需要内存空间。为了后面的代码访问LDT，LDT 的线性基地址要登记到任务控制块（TCB）中。

第661～667 行，创建用户任务的代码段描述符，并登记到LDT 中。从程序中可见，代码段描述符中的基地址是0x00000000，段界限值是0x000FFFFF，粒度为4KB，因此，实际使用的界限值是0xFFFFFFFF。用户任务的特权级别是3，因此，代码段描述符的特权级别也是3，段选择子的特权级别也设置成3。

第669、670 行，将代码段描述符的选择子登记到任务状态段（TSS）中。TSS 的线性地址是从任务控制块（TCB）中取得的。

第673 ～ 679 行， 创建用户任务的数据段描述符。和代码段描述符一样， 基地址为0x00000000，段界限也是0x000FFFFF，粒度为4KB。描述符特权级和段选择子的特权级都是3。

在平坦模型下，段寄存器DS、ES、FS 和GS 都指向同一个4GB 数据段。因此，第681～685行，将刚才生成的数据段描述符选择子填写到TSS 的DS、ES、FS 和GS 寄存器域中。

在平坦模型下，段只是容器，很大的容器。之所以要将段定义成4GB 大小，是希望可以发出任何虚拟地址，而不会被段部件的检查机制阻挠。当然了，骗得过段部件，骗不了页部件。容器是很大，但是，能

不能拿到东西，要看你手伸到的地方有没有东西。因此，访问一个虚拟内存位置之前，必须提前在那里分配页。

在平坦模型下，栈段也要和其他段共享4GB 的虚拟内存空间。用户任务的数据段是3 特权级别的，而该任务固有的栈也是3 特权级别的，可以把上面的数据段选择子赋给段寄存器SS，把数据段作为栈段来用。真的可以吗？答案是，完全可以。尽管一般来说数据段是向上扩展的，而栈段是向下扩展的，但是，向上和向下，并不是用来限制压栈和出栈操作，而是规定处理器段部件检查段界限的方法。如果把向上扩展的数据段作为栈来用，那么，每当执行隐式的栈操作指令时（push、pop、call、ret 和iret），处理器的段部件按向上扩展的段来检查段界限，指令的执行过程和栈的推进方向依然不变，是向低地址方向的。

我们说了，段是容器，有4GB 大小，但必须提前分配页给那些要访问到的地方才行。在我们的用户程序（代码清单16-2）中，仅有数据和代码，没有定义栈空间。所以，定义了4GB 的栈段后，还要分配实际的栈空间才行。第688～690 行，在用户任务自己的虚拟地址空间内分配内存，分配了4KB，所以用户任务的固有栈就是4KB。第692～695 行，将CX 寄存器中的数据段选择子填写到TSS 的SS 寄存器域中，同时，填写TSS 的ESP 寄存器域。由于栈从内存的高端向低端推进，所以，ESP 寄存器域的内容被指定为TCB 中的下一个可分配的线性地址。

接下来是创建0、1、2 特权级的栈。毫无疑问，这三个栈段的基地址也是0x00000000，也要创建为向上扩展的数据段，段界限为0x000FFFFF，粒度为4KB。当然，这三个栈段，其描述符的特权级别不同，段选择子也不一样。第698～749 行就是用来创建这三个栈段的代码的，很好懂，不再一一解释。

截至现在，用户程序已经加载，相关的段描述符已经创建，如图16-26 所示。

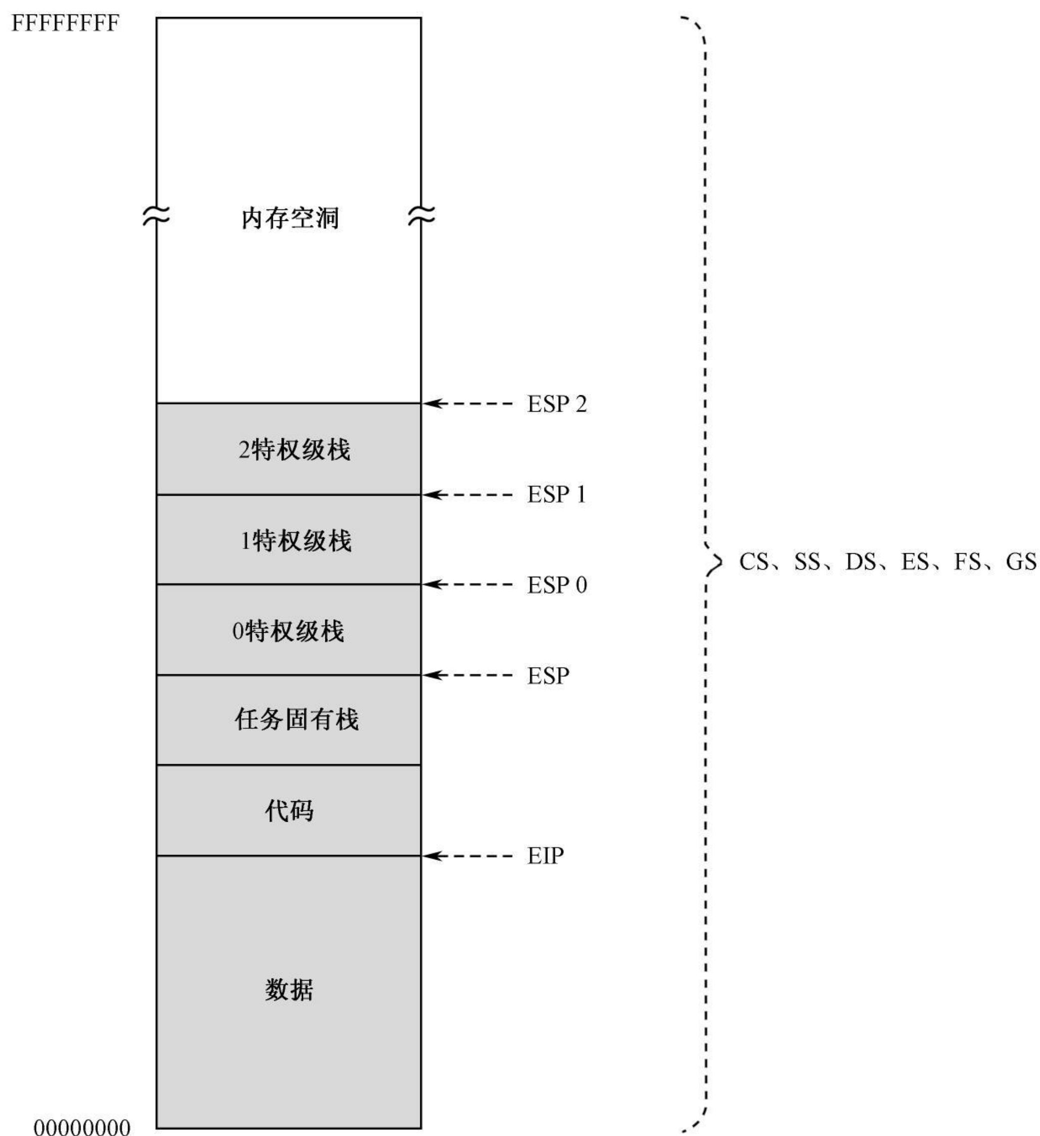


图16-26　用户任务的虚拟内存空间布局图

16.5.6　重定位U-SALT 并复制页目录表

又到了重定位SALT 表的时候了，第753～794 行是重定位SALT 的代码。这段程序不管是在哪一章，都只有两行不一样，其余都一模一样，没有变化。具体到本章中，这两行是

```
761         mov ecx,[es:0x0c]         ;U-SALT 条目数
762         mov edi,[es:0x08]         ;U-SALT 在 4GB 空间内的偏移
```

由于使用了平坦模型，而且启动了分页功能，所以，可以直接访问用户程序的虚拟内存空间，从中取得SALT 表的条目数和线性地址（4GB 段内偏移量）。

第797～803 行，创建LDT 描述符，并登记在GDT 中。处理器要求LDT 描述符必须登记在GDT 中。

第805～820 行，填写任务状态段（TSS）的其余部分。包括LDT 选择子域、I/O 位映射区的偏移地址、前一任务的TSS 链接域、TSS 的界限、EIP 和EFLAGS 寄存器域。特别要提到的是，EIP 域填写的是用户程序的入口点，这很重要，从内核任务切换到用户任务时，是用TSS 中的内容恢复现场的，所以这关系到任务应该从哪里开始执行。EFLAGS 域的内容是当前内核任务EFLAGS 寄存器的副本。

第823～828 行，创建TSS 描述符，并登记到GDT 中。处理器要求TSS 描述符必须登记在GDT 中。TSS 描述符的特权级DPL 必须是0，只有当前特权级别为0 的程序才能转换到该任务。因为任务切换应当由内核发起，而特权级为1、2 或3 的程序一般不允许主动发起任务切换。

现在，几乎所有的工作都完成了，剩下的事情，就是创建属于用户任务自己的页目录表。毕竟，你只是临时借用了内核任务的页目录表，当用户任务真正开始执行时，它不可能还使用内核的页目录表，那太不像话了。

我们说过，创建用户任务时，使用内核的页目录表，然后，再复制它，作为用户任务的页目录表。页目录表的复制工作是调用过程create_copy_cur_pdir 完成的。该过程位于第406 行，属于公共例程段。

第416～418 行，先令段寄存器DS 和ES 都指向0～4GB 数据段。说实话，如果内核也工作在平坦模型下，就不用这么麻烦了。

要创建页目录表，那当然先得分配一个空闲页。第420 行，首先调用过程allocate_a_4k_page分配一个页，页的物理地址由EAX 寄存器返回。第422 行，将页物理地址的低12 位改成属性，属性值为0x007，即，US＝1，允许特权级别为3 的用户程序访问该页；RW＝1，页是可读可写的；P＝1，页位于物理内存中。

为了能够访问到该页，我们把它的物理地址登记到当前页目录表的倒数第2 个目录项。我们知道，当前页目录表的线性地址是0xFFFFF000，它的倒数第2 个目录项在表内的偏移量为0xFF8。因此，

页目录表内倒数第2 个目录项的线性地址是0xFFFFFFF8，第423 行，将附加了属性的页地址登记到该目录项。

要访问这个新的页目录表，必须知道它的线性地址。事实上，它的线性地址是0xFFFFE000。事情是这样的，让我们倒过来分析一下。首先，线性地址0xFFFFE000 的页目录索引值是0x3FF，指向当前页目录表的最后一项；页表索引值为0x3FE，是页表内倒数第2 项，存放的是页地址。要知道，当前页目录的最后一个目录项，存放的是页目录表自己的物理地址，页表就是当前页目录表自己，而倒数第2 个目录项，又是新页目录表的物理地址。这就证明了，0xFFFFE000 的确是新页目录表的线性地址。

既然两个表的线性地址都有了，那么，使用带rep 前缀的movsd 指令做表间复制工作最为方便。第425～429 行，按处理器的要求，设置好ESI 和EDI 寄存器，分别令它们指向当前页目录表和新页目录表；设置ECX 寄存器的内容为传送的次数（目录项数）；cld 指令设置传送的方向为正向，即，ESI 和EDI 在每次传送后递增。最后，执行movsd 指令，自动进行复制工作。

复制工作完成后，控制返回到第833 行。在那里，将新页目录表的物理地址填写到用户任务TSS 的CR3 寄存器域中。

最后，ret 8 指令从栈中弹出参数，并返回到第1069 行。

在多任务环境下，可以创建多个用户任务，使它们同时运行。如图16-27 所示，每个任务都拥有自己独立的4GB 虚拟地址空间，而且互相隔离。其中，前2GB 属于任务的私有地址空间，高2GB 是所有任务共有的全局地址空间，映射到内核的地址空间内。

每个任务都有自己独立的页目录表和页表，页目录表的前半部分对应着任务虚拟地址空间的前2GB，后半部分则映射到内核的页表。这样，当任务在自己独立的局部空间工作时，使用它自己的页表；当任务请求系统服务时，用的则是内核的页表，访问的是内核的代码和数据。

我相信，用这幅图作为最后的总结，可以使读者更清楚地在脑海中勾勒出分页机制下的虚拟内存分配全景，进一步加深对多任务、分页和虚拟内存分配原理的理解。

16.5.7　切换到用户任务执行

第1069～1072 行，先显示一条消息，告诉屏幕前的人，正在执行任务切换。接着，使用call指令发起任务切换。call 指令的参数是TCB 中的TSS 选择子。任务切换时，内核任务的状态被保存到当前的TSS 中，接着，找到用户任务的TSS，从中取出各种参数，加载到处理器的各个寄存器中，包括CR3 寄存器、LDTR、段寄存器、指令指针和栈指针寄存器、通用寄存器等。于是，用户任务就开始执行了。

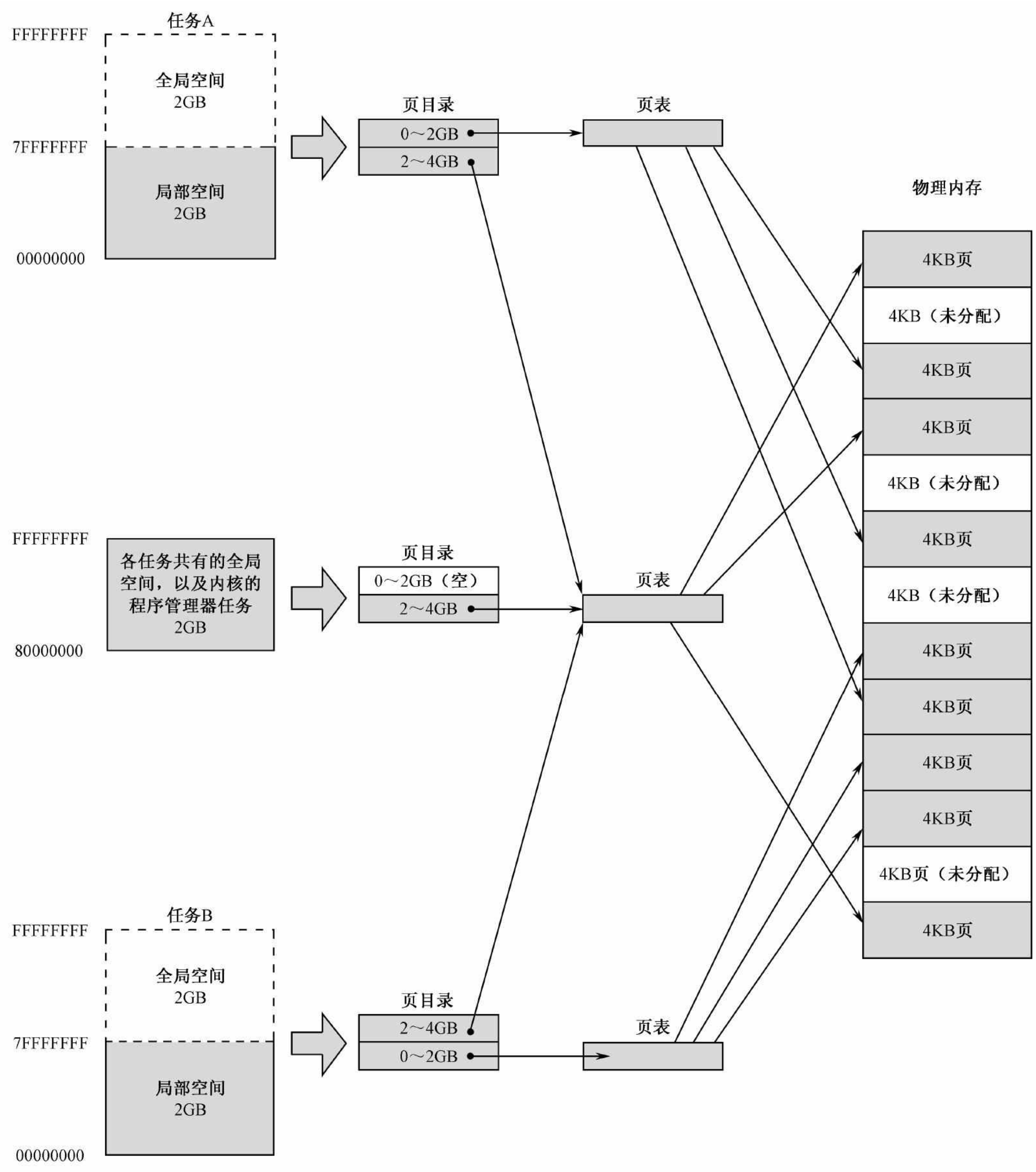


图16-27　多任务环境下的页目录表和页表映射示意图

来看代码清单16-2。

第46、47 行，显示字符串，表示任务当前正工作在页功能开启的状态下。

接着，第49～59 行，以十六进制的形式，显示当前任务4GB 虚拟地址空间内的前88 个双字。每次先显示两个空格，然后再显示双字的值，这样形成的效果是每行8 个双字，共11 行。如图16-28 所示，这是当前任务的运行结果。

空格字符串在第38 行，用标号space 声明，并初始化为3 字节。空格的ASCII 码为0x20，因此，这一行等效于

```
        space  db ’  ’,0
```

结合代码清单16-2，再来看屏幕上显示的内容。0x0001F88E 是用户程序的总长度，即129166字节；第2 个双字是0x1F85B，是用户程序的入口点；第3 个双字是0x00000010，这是U-SALT表的起始线性地址；第4 个双字是0x000001F8，这是U-SALT 表的条目数。不要忘了，我们在SALT 表的中间插入了以下语句：

```
        reserved  times 256*500 db 0      ;保留一个空白区，以演示分页
```

这直接导致多出500 个表项。加上原有的4 个表项，共504（0x000001F8）个表项。

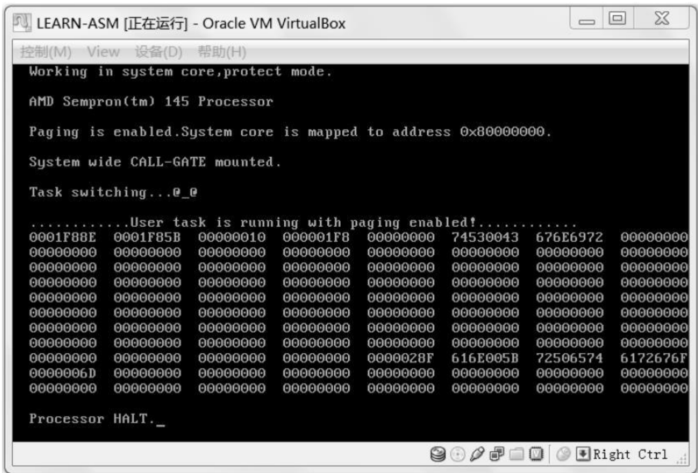


图16-28　本章程序的运行结果截图

从源程序中可知，紧接着显示的是U-SALT 表的内容。不过，每个表项的前6 字节已经被内核改成调用门选择子和偏移量了。SALT 表的内容是按字节定义的，当按双字值显示在屏幕上时，会显得颠倒错乱，请读者在有时间的情况下自行分析。

第61 行，将控制返回到内核中。

回到代码清单16-1。

控制返回到代码清单16-1 的第445 行。在那里，先根据EFLAGS 寄存器的NT 位，判断用户任务（当前任务）当初是怎么发起的。如果是由call 指令发起的，那么，就用iretd 指令转换到前一个任务（内核）；如果是由jmp 指令发起的，则直接用jmp 指令转换回内核任务。

无论如何，当任务切换后，一定会转换回内核任务，因为当前就两个任务。一旦内核任务恢复执行，而且执行点在第1074 行，紧接着当初发起任务切换的那条指令之后。

第1074～1077 行，内核显示一条消息，表示处理器要停机了。于是，它说到做到，停机。

16.6　程序的编译、执行和调试

16.6.1　本章程序的编译和运行方法

分别编译代码清单16-1（c16_core.asm）和16-2（c16.asm），并将编译后的文件写入虚拟硬盘，前者从逻辑扇区1 开始写入，后者从逻辑扇区50 开始写入。完成以上两项工作后，启动VirtualBox，就可以观察到运行结果。

16.6.2　察看CR3 寄存器的内容

可以在向页目录基地址寄存器PDBR（即控制寄存器CR3）写入页目录物理地址后察看它的内容。方法我们前面讲过，就是使用Bochs 的调试命令“creg”。

如图16-29 所示，这是在执行了代码清单16-1 中以下指令后的控制寄存器状态：

```
mov eax,0x00020000                      ;PCD=PWT=0
mov cr3,eax

mov eax,cr0
or eax,0x80000000
mov cr0,eax                            ;开启分页机制
```

从图中可以清楚地看到，由于已经处于分页模式下（必须要先处于保护模式），所以CR0 的PE 位和PG 位都已处于置位状态，控制寄存器CR3 中的内容是当前任务页目录的物理地址，即，0x20000，PCD=0，页级缓存被禁用；PWT=0，页级通写被禁用。

```
Bochs for Windows - Console
(0) [0x0000000000040ecb] 0038:000000000000048b (unk. ctxt): mov eax, cr0
      ; 0f20c0
<bochs:157> n
Next at t=17847105
(0) [0x0000000000040ece] 0038:000000000000048e (unk. ctxt): or eax, 0x80000000
      ; 0d00000080
<bochs:158> n
Next at t=17847106
(0) [0x0000000000040ed3] 0038:0000000000000493 (unk. ctxt): mov cr0, eax
      ; 0f22c0
<bochs:159> s
Next at t=17847107
(0) [0x0000000000040ed6] 0038:0000000000000496 (unk. ctxt): mov ebx, 0xfffff000
      ; bb00f0ffff
<bochs:160> creg
CR0=0xe0000011: PG CD NW ac wp ne ET ts em mp PE
CR2=page fault laddr=0x0000000000000000
CR3=0x0000000000020000
    PCD=page-level cache disable=0
    PWT=page-level write-through=0
CR4=0x00000000: smep osxsave pcid fsgsbase smx vmx osxmmexcpt osfxsr pce pge mce
 pae pse de tsd pvi vme
CR8: 0x0
EFER=0x00000000: ffxsr nxe lma lme sce
<bochs:161> _
```

图16-29　开启了分页模式后的控制寄存器状态

16.6.3　察看线性地址对应的物理页信息

一旦进入分页模式，可以用“page”命令察看线性地址到物理页的映射信息。比如，如图16-30 所示，这里显示了线性地址0x7e08 所对应的物理页信息。

```
Bochs for Windows - Console
rax: 0x00000000_e0000011 rcx: 0x00000000_00000000
rdx: 0x00000000_000ff003 rbx: 0x00000000_00021000
rsp: 0x00000000_00000000 rbp: 0x00000000_00000000
rsi: 0x00000000_00000400 rdi: 0x00000000_00040000
r8 : 0x00000000_00000000 r9 : 0x00000000_00000000
r10: 0x00000000_00000000 r11: 0x00000000_00000000
r12: 0x00000000_00000000 r13: 0x00000000_00000000
r14: 0x00000000_00000000 r15: 0x00000000_00000000
rip: 0x00000000_00000496
eflags 0x00000086: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af PF cf
<bochs:96>
CR0=0xe0000011: PG CD NW ac wp ne ET ts em mp PE
CR2=page fault laddr=0x0000000000000000
CR3=0x0000000000020000
    PCD=page-level cache disable=0
    PWT=page-level write-through=0
CR4=0x00000000: smep osxsave pcid fsgsbase smx vmx osxmmexcpt osfxsr pce pge mce
 pae pse de tsd pvi vme
CR8: 0x0
EFER=0x00000000: ffxsr nxe lma lme sce
<bochs:97> page 0x7e08
 PDE: 0x0000000000021003    ps         a pcd pwt S W P
 PTE: 0x0000000000007003       g pat d a pcd pwt S W P
linear page 0x0000000000007000 maps to physical page 0x0000000000007000
<bochs:98>
```

图16-30　察看线性地址到物理页的映射信息

如图中所示，与线性地址0x7e08 对应的页表，其物理地址登记在页目录表中，是作为页目录项PDE 存在的，该目录项PDE 的内容是0x21003。即，页表的物理地址是0x21000。与线性地址0x7e08 对应的物理页，其地址登记在页表中，是作为页表项PTE 存在的，该页表项PTE 的内容是0x7003。这就是说，与线性地址0x7e08 相对应的页是0x7000。

16.6.4 察看当前任务的页表信息

可以察看当前任务的页表，显示线性地址和物理地址（页）的全部映射关系。要做到这一点，可以使用Bochs 的调试命令“info tab”。如图16-31 所示，这是在本章中初次进入分页模式后，页表的信息。

如图中所示，虚拟内存空间的低端1MB，即线性地址0x00000000～0x000FFFFF，对应着物理地址0x00000000～0x000FFFFF。这是可以理解的，因为在初次进入分页模式时，我们需要建立这低端1MB 内存空间的一一映射，使线性地址和物理地址相同。

```
Bochs for Windows - Console
Next at t=17847106
(0) [0x000000000040ed3] 0038:0000000000000493 (unk. ctxt): mov cr0, eax
      ; 0f22c0
<bochs:101> n
Next at t=17847107
(0) [0x000000000040ed6] 0038:0000000000000496 (unk. ctxt): mov ebx, 0xfffff000
      ; bb00f0ffff
<bochs:102> help info
info break - show information about current breakpoint status
info idt - show interrupt descriptor table
info ivt - show interrupt vector table
info gdt - show global descriptor table
info tss - show current task state segment
info tab - show page tables
info eflags - show decoded EFLAGS register
info symbols [string] - list symbols whose prefix is string
info device - show list of devices supported by this command
info device [string] - show state of device specified in string
info device [string] [string] - show state of device with options
<bochs:103> info tab
cr3: 0x0000000000020000
0x00000000-0x000fffff -> 0x0000000000000000-0x00000000000fffff
0xffc00000-0xffc00fff -> 0x0000000000021000-0x0000000000021fff
0xfffff000-0xffffffff -> 0x0000000000020000-0x0000000000020fff
<bochs:104> _
```

图16-31　初次进入分页模式后的页表信息

为了用线性地址来修改页表的内容，我们把页表当成普通的页，把页目录表当成页表来用。在这种情况下，页表的线性首地址是0xFFC00000。因此，0xFFC00000～0xFFC00FFF 这段4KB的线性地址区间对应的是页表的实际物理地址0x00021000～0x00021FFF。

为了用线性地址访问和修改页目录表自己，页目录表的最后一个目录项，登记的是页目录表自己的物理地址。因此，页目录表的线性地址是0xFFFFF000。即，0xFFFFF000～0xFFFFFFFF 这段4KB 线性地址区间对应着实际的物理地址区间0x00020000～0x00020FFF。

在本章中，一旦通过单步执行，进入用户程序执行后，就可以察看用户任务的页表信息。如图16-32 所示，用户程序的页表信息比较庞大，但非常清楚地显示了整个内存空间的映射关系，特别是全局空间（内核）和局部空间是如何映射的。

```
Bochs for Windows - Console
0x00018000-0x00018fff -> 0x0000000000127000-0x0000000000127fff
0x00019000-0x00019fff -> 0x0000000000129000-0x0000000000129fff
0x0001a000-0x0001afff -> 0x000000000012b000-0x000000000012bfff
0x0001b000-0x0001bfff -> 0x000000000012d000-0x000000000012dfff
0x0001c000-0x0001cfff -> 0x000000000012f000-0x000000000012ffff
0x0001d000-0x0001dfff -> 0x0000000000131000-0x0000000000131fff
0x0001e000-0x0001efff -> 0x0000000000133000-0x0000000000133fff
0x0001f000-0x0001ffff -> 0x0000000000135000-0x0000000000135fff
0x00020000-0x00020fff -> 0x0000000000139000-0x0000000000139fff
0x00021000-0x00021fff -> 0x000000000013b000-0x000000000013bfff
0x00022000-0x00022fff -> 0x000000000013d000-0x000000000013dfff
0x00023000-0x00024fff -> 0x000000000013f000-0x0000000000140fff
0x80000000-0x800fffff -> 0x0000000000000000-0x00000000000fffff
0x80100000-0x80100fff -> 0x0000000000029000-0x0000000000029fff
0x80101000-0x80101fff -> 0x000000000002b000-0x000000000002bfff
0x80102000-0x80102fff -> 0x0000000000137000-0x0000000000137fff
0xff800000-0xff800fff -> 0x000000000002d000-0x000000000002dfff
0xffa00000-0xffa00fff -> 0x0000000000021000-0x0000000000021fff
0xffbfe000-0xffbfefff -> 0x0000000000141000-0x0000000000141fff
0xffbff000-0xffbfffff -> 0x0000000000020000-0x0000000000020fff
0xffc00000-0xffc00fff -> 0x000000000002d000-0x000000000002dfff
0xffe00000-0xffe00fff -> 0x0000000000021000-0x0000000000021fff
0xffffe000-0xffffefff -> 0x0000000000141000-0x0000000000141fff
0xfffff000-0xffffffff -> 0x0000000000020000-0x0000000000020fff
<bochs:6>
```

图16-32　本章用户任务的页表信息

现在，请你根据本章中用户程序的结构，结合Bochs 所显示的页表信息，逐一解释这些对应关系的含义。

16.6.5　使用线性（虚拟）地址调试程序

开启了分页模式之后，程序开始使用线性地址工作。相应地，在调试程序时，有时候只能知道线性地址，而不知道物理地址。在这种情况下，Bochs 允许你使用线性地址工作。

举个例子，我们以前一直用"xp"命令来显示内存单元的内容。相应地，"x"命令则显示一个线性地址单元里的内容。如图16-33 所示，当本章的用户任务开始执行后，它拥有独立的4GB虚拟地址空间。因此，我们可以用"x 0x0"命令显示当前任务4GB 虚拟空间内的头一个双字。

```
Bochs for Windows - Console
0x0001f000-0x0001ffff -> 0x0000000000135000-0x0000000000135fff
0x00020000-0x00020fff -> 0x0000000000139000-0x0000000000139fff
0x00021000-0x00021fff -> 0x000000000013b000-0x000000000013bfff
0x00022000-0x00022fff -> 0x000000000013d000-0x000000000013dfff
0x00023000-0x00024fff -> 0x000000000013f000-0x0000000000140fff
0x80000000-0x800fffff -> 0x0000000000000000-0x00000000000fffff
0x80100000-0x80100fff -> 0x0000000000029000-0x0000000000029fff
0x80101000-0x80101fff -> 0x000000000002b000-0x000000000002bfff
0x80102000-0x80102fff -> 0x0000000000137000-0x0000000000137fff
0xff800000-0xff800fff -> 0x000000000002d000-0x000000000002dfff
0xffa00000-0xffa00fff -> 0x0000000000021000-0x0000000000021fff
0xffbfe000-0xffbfefff -> 0x0000000000141000-0x0000000000141fff
0xffbff000-0xffbfffff -> 0x0000000000020000-0x0000000000020fff
0xffc00000-0xffc00fff -> 0x000000000002d000-0x000000000002dfff
0xffe00000-0xffe00fff -> 0x0000000000021000-0x0000000000021fff
0xffffe000-0xffffefff -> 0x0000000000141000-0x0000000000141fff
0xfffff000-0xffffffff -> 0x0000000000020000-0x0000000000020fff
<bochs:6> s
Next at t=18215716
(0) [0x0000000000135860] 0007:000000000001f860 (unk. ctxt): call far ds:0x10
      ; ff1d10000000
<bochs:7> x 0x0
[bochs]:
0x0000000000000000 <bogus+       0>:    0x0001f88e
<bochs:8>
```

图16-33　使用线性地址显示内存数据

对照本章代码16-2，线性地址为0 的地方，是程序的总大小，在这里是0x0001f88e。即，本章用户程序的大小是129166 字节。

除了在显示内存数据的时候使用线性地址，也可以在知道指令线性地址的时候，用线性地址设置断点，其命令是“lb”或者“vb”。具体使用方法，请使用Bochs 的帮助命令“help”（help lb、help vb）。

本 章 习 题

1．代码清单16-2（c16.asm）的第47 行是通过调用门进入系统核心显示字符串的指令：

```
        47     call far [PrintString]
```

请以该指令的执行过程为例，说明为什么必须将系统核心映射到每个任务的4GB 地址空间内才行？

2．修改代码清单16-2（c16.asm），显示当前任务前50 个页面的物理地址，可以使用现成的调用门。提示：必须在代码清单16-1（c16_core.asm）中修改页目录的访问属性。

3．（选做）如果可能的话，尝试修改代码清单16-1，使系统核心工作在平坦模式下。

第17章　中断和异常的处理与抢占式多任务

说实话，在开始这一章的写作任务之前，我从来没有意识到它这么重要。相反，我以为它可以很轻松。

一开始，在我的规划中，还有第18 章，名字我都想好了，叫“抢占式的多任务轮转和调度”，而且相信这样的名字会让很多喜欢钻研的人感兴趣。

但是，很快我就意识到，这两章的内容都很单一，而且这些内容互相交叉，联系很紧密。想想看，抢占式的多任务轮转，是离不开中断的。因此，我决定把原定的两章合为一章，就是这一章。

中断和异常中断的内容是很重要的。至少，你需要知道处理器会引发哪些异常，在什么情况下会引发异常，这对于编写正确的程序，以及调试这些程序来说，是非常重要的。但是，还记得上一章的习题吗？有一道题是选做的，要求用平坦内存模型来改写系统内核，我突然想到这是一道非常难的习题，也许没有一个初学者能够完成这个任务。进而我又想到，如果我把这一章的内核代码写成平坦内存模型的，将是一件了不起的事。之所以了不起，是因为它可以使读者们更加深入地，不，是彻底地理解分页机制的原理。于是，我花了一天的时间，终于把内核代码改成了平坦模型。

然后，我又发现，作为一本汇编语言教程，宏（Macro）是不可或缺的。绝大多数的汇编语言编译器都支持宏，绝大多数的教科书都要着重地讲到宏。因此，不讲宏，对不起读者，对不起汇编语言。

穿过这一章，这本书就结束了，让我们开始吧。本章的目标是：

1．　了解保护模式下x86 处理器中断和异常中断的工作机制，知道中断和异常中断的分类，认识中断描述符表IDT、中断门和陷阱门。

2．　使本章程序完全在平坦内存模型下工作，进一步加深对分页机制、TLB 以及虚拟地址空间等概念的理解，了解在平坦内存模型下进行汇编语言程序设计的特点。

3． 使用硬件中断实施抢占式多任务切换，彻底理解任务切换的原理和过程。在这个过程中，学习一种简单的任务调度算法，以及单向链表的遍历、节点的追加／插入／删除／移动算法。

4． 学习宏汇编技术的应用。

5． 学习几条新的x86 处理器指令，包括lidt、invlpg、bound 和ud2 等。

17.1 中断和异常

17.1.1 中断和异常概述

你应该对中断并不陌生，毕竟我们已经学习过它的知识，也用它来写过程序。中断和异常的作用是指示系统中的某个地方发生了一些事件，需要引起处理器（包括正在执行中的程序和任务）的注意。当中断和异常发生时，典型的结果是迫使处理器将控制从当前正在执行的程序或任务转移到另一个例程或者任务中去。该例程叫做中断处理程序，或者异常处理程序。如果是一个任务，则发生任务切换。

1．中断（Interrupt）

中断包括硬件中断和软中断。

硬件中断是由外围硬件设备发出的中断信号引发的，以请求处理器提供服务。当I/O 接口发出中断请求时，会被像8259A 和I/O APIC 这样的中断控制器收集，并发送到处理器。硬件中断完全是随机产生的，与处理器的执行并不同步。当中断发生时，处理器要先执行完当前的指令，然后才对中断进行处理。

软中断是由int n 指令引发的中断处理，n 是中断号或者叫类型码。

2．异常（Exception）

异常就是我们在介绍16 位汇编语言时所说的内部中断。它们是处理器内部产生的中断，表示在指令执行的过程中遇到了错误的状况。当处理器执行一条非法指令，或者因条件不具备，指令不能正常执行时，将引发这种类型的中断。以上所列的情况都是异常情况，所以内部中断又叫异常或者异常中断。比如，在执行除法指令div/idiv 时，遇到了被0 除的情况（除数是0）；再比如，使用jmp 指令发起任务切换时，指令的操作数不是一个有效的TSS 描述符选择子。

异常分为三种，第一种是程序错误异常，指处理器在执行指令的过程中，检测到了程序中的错误，并由此而引发的异常。

第二种是软件引发的异常。这类异常通常由into、int3 和bound 指令主动发起。这些指令允许在指令流的当前点上检查实施异常处理的条件

是否满足。举个例子来说，into 指令在执行时，将检查EFLAGS 寄存器的OF 标志位，如果满足为“1”的条件，则引发异常。

第三种是机器检查异常。这种异常是处理器型号相关的，也就是说，每种处理器都不太一样。无论如何，处理器提供了一种对硬件芯片内部和总线处理进行检查的机制，当检测到有错误时，将引发此类异常。

根据异常情况的性质和严重性，异常又分为以下三种，并分别实施不同的处理。

●故障（Faults）。故障通常是可以纠正的，比如，当处理器执行一个访问内存的指令时，发现那个段或者页不在内存中（P＝0），此时，可以在异常处理程序中予以纠正（分配内存，或者执行磁盘的换入换出操作），返回时，程序可以重新启动并不失连续性。为了做到这一点，当故障发生时，处理器把机器状态恢复到引起故障的那条指令之前的状态，在进入异常处理程序时，压入栈中的返回地址（CS 和EIP 的内容）是指向引起故障的那条指令的，而不像通常那样指向下一条指令。如此一来，当中断返回时，将重新执行引起故障的那条指令，而且不再出错（如果引起异常的情况已经妥善处置）。这意味着，异常并不总是意味着坏消息，相反，很多时候，它是有益的，就像益虫。如果没有异常，虚拟内存管理将无从谈起。

●陷阱（Traps）。陷阱中断通常在执行了截获陷阱条件的指令之后立即产生，如果陷阱条件成立的话。陷阱通常用于调试目的，比如单步中断指令int3 和溢出检测指令into。陷阱中断允许程序或者任务在从中断处理过程返回之后继续进行而不失连续性。因此，当此异常发生时，在转入异常处理程序之前，处理器在栈中压入陷阱截获指令的下一条指令的地址。

●终止（Aborts）。终止标志着最严重的错误，诸如硬件错误、系统表（GDT、LDT 等）中的数据不一致或者无效。这类异常总是无法精确地报告引起错误的指令的位置，在这种错误发生时，程序或者任务都不可能重新启动。一个比较典型的终止类异常是“双重故障”（中断号为8），当发生一次异常后，处理器在转入该中断的处理程序时，又发生另外的异常（如该中断处理程序所在的段不在内存中，或者栈溢出）。对于中断处理程序来说，很难从栈中获得有关如何纠正此类错误的明确信息，往往是发生极为重大的错误时才伴随着这种异常，所以再继续执行

引起此异常的程序或任务已相当困难，操作系统通常只能把该任务从系统中抹去。

中断和异常发生时，处理器将挂起当前正在执行的过程或者任务，然后执行中断和异常处理过程。返回时，处理器恢复程序或者任务的执行，而且被打断的程序或任务的执行不失连续性，除非遇到一个终止类型的异常。对于某些异常，处理器在转入异常处理程序之前，会在当前栈中压入一个称为错误代码的数值，帮助程序进一步诊断异常产生的位置和原因。

表17-1 列出了Intel 处理器在保护模式下的中断和异常。

表17-1　保护模式下的中断和异常向量分配

向量	助记	描　述	类型	错误代码	来　源
0	#DE	除法错	故障	无	div 或 idiv 指令
1	#DB	保留			
2	-	NMI	中断	无	不可屏蔽的外部中断
3	#BP	断点	陷阱	无	int3 指令
4	#OF	溢出	陷阱	无	into 指令
5	#BR	对数组的引用超出边界	故障	无	bound 指令
6	#UD	无效或未定义的操作码	故障	无	ud 2 指令，或保护的操作码
7	#NM	设备不可用（无数学协处理器）	故障	无	浮点或者 wait/fwiat 指令
8	#DF	双重故障	终止	有（0）	任何会产生异常的指令、NMI 或者硬件中断
9		协处理器段超越（保留）。协处理器执行浮点运算时，至少有两个操作数不在一个段内（跨段）	故障	无	浮点指令
10	#TS	无效 TSS	故障	有	任务切换或访问 TSS
11	#NP	段不存在	故障	有	加载段寄存器或者访问系统段
12	#SS	栈段故障	故障	有	栈操作或者加载段寄存器 SS
13	#GP	常规保护	故障	有	任何内存引用或其他保护检查
14	#PF	页故障	故障	有	任何内存引用
15	-	由 Intel 处理器保留，不能使用		无	
16	#MF	x87 FPU（浮点处理单元）浮点处理错误	故障	无	x87 FPU 浮点指令或 Wait/Fwiat 指令
17	#AC	对齐检查	故障	有（0）	任何内存数据引用
18	#MC	机器检查	终止	无	错误代码（如果有的话）和来源是处理器型号相关的
19	#XM	SIMD（单指令多数据）浮点异常	故障	无	sse/sse 2/sse 3 浮点指令
20～31		Intel 公司保留，建议不要使用			
32～255		用户自定义的中断	中断		外部中断，或者 int n 指令

当中断和异常发生时，NMI 和异常的向量是由处理器自动给出的；硬件中断的向量是由I/O中断控制器芯片送给处理器的；软中断的向量是由指令中的操作数给出的。

从80486 之后开始，处理器内部一般集成了浮点运算部件x87 FPU，不再需要安装独立的数学协处理器，所以有些和浮点运算有关的异常可能不会产生（比如向量为9 的协处理器段超越故障）。wait 和fwait 指令用于主处理器和浮点处理部件（FPU）之间的同步，它们应当放在浮点指令之后，以捕捉任何浮点异常。

从1993 年的Pentium 处理器开始，引入了用于加速多媒体处理的多媒体扩展技术（Multi-Media eXtension，MMX），该技术使用单指令多数据（Single-Instruction，Multiple-Data，SIMD）执行模式，以便于在64 位的寄存器内实施并行的整数计算。在之后的岁月里，随着处理器的更新换代，该项技术也多次扩展，第一次扩展被称为SSE（SIMD Extension），第二次是SSE2，第三次是SSE3。和SIMD 有关的异常是从Pentium III处理器开始引入的。

bound（Check Array Index Against Bounds）指令用于检查数组的索引是否在边界之内，其格式为

```
bound r16,m16
bound r32,m32
```

其具有两个操作数，目的操作数是寄存器，包含了数组的索引；源操作数必须指向内存位置，在那里包含了两个成对出现的字或者双字，分别是数组索引的下限和上限。如果执行bound指令时，数组的索引小于下标的下限，或者大于下标的上限，则产生异常。

ud2（Undefined Instruction）指令是从Pentium Pro 处理器开始引入的，它只有操作码而没有操作数，机器代码为0F 0B。

```
ud2            ;机器码 0F 0B
```

执行该指令时，会引发一个无效操作码异常。该指令没有别的用处，典型地用于软件测试。尽管异常是用该指令故意引发的，但是，在转入异常处理程序时，压入栈中的指令指针是指向该指令的，而非下一条指令。

17.1.2　中断描述符表、中断门和陷阱门

在实模式下，位于内存最低端的1KB 内存，是中断向量表（IVT），定义了256 种中断的入口地址，包括16 位段地址和16 位段内偏移量。当中断发生时，处理器要么自发产生一个中断向量，要么从int n 指令中得到中断向量，或者从外部的中断控制器接受一个中断向量。然后，它将该向量作为索引访问中断向量表。具体的做法是，将中断向量乘以4，作为表内偏移量访问中断向量表，从中取得中断处理过程的段地址和偏移地址，并转到那里执行。

在保护模式下，处理器对中断的管理是相似的，但并非使用传统的中断向量表来保存中断处理过程的地址，而是中断描述符表（Interrupt Descriptor Table，IDT）。顾名思义，在这个表里，保存的是和中断处理过程有关的描述符，包括中断门、陷阱门和任务门。

任务门的格式我们已经在前面的章节里介绍过了，中断门和陷阱门的格式如图17-1 所示。

事实上，调用门、任务门、中断门和陷阱门的描述符非常相似，从大的方面来说，因为都用于实施控制转移，故都包括16 位的目标代码段选择子，以及32 位的段内偏移量。由图17-1 中可见，中断门和陷阱门仅仅有一比特的差别。中断门和陷阱门描述符只允许存放在IDT 内，任务门可以位于GDT、LDT 和IDT 中。

和实模式下的中断向量表（IVT）不同，保护模式下的IDT 不要求必须位于内存的最低端。事实上，在处理器内部，有一个48 位的中断描述符表寄存器（Interrupt Descriptor Table Register，IDTR），保存着中断描述符表在内存中的线性基地址和界限。如图17-2 所示，和GDT 一样，因为整个系统中只需要一个IDT 就够了，所以，GDTR 与IDTR 不像LDTR 和TR，没有也不需要选择器部分。

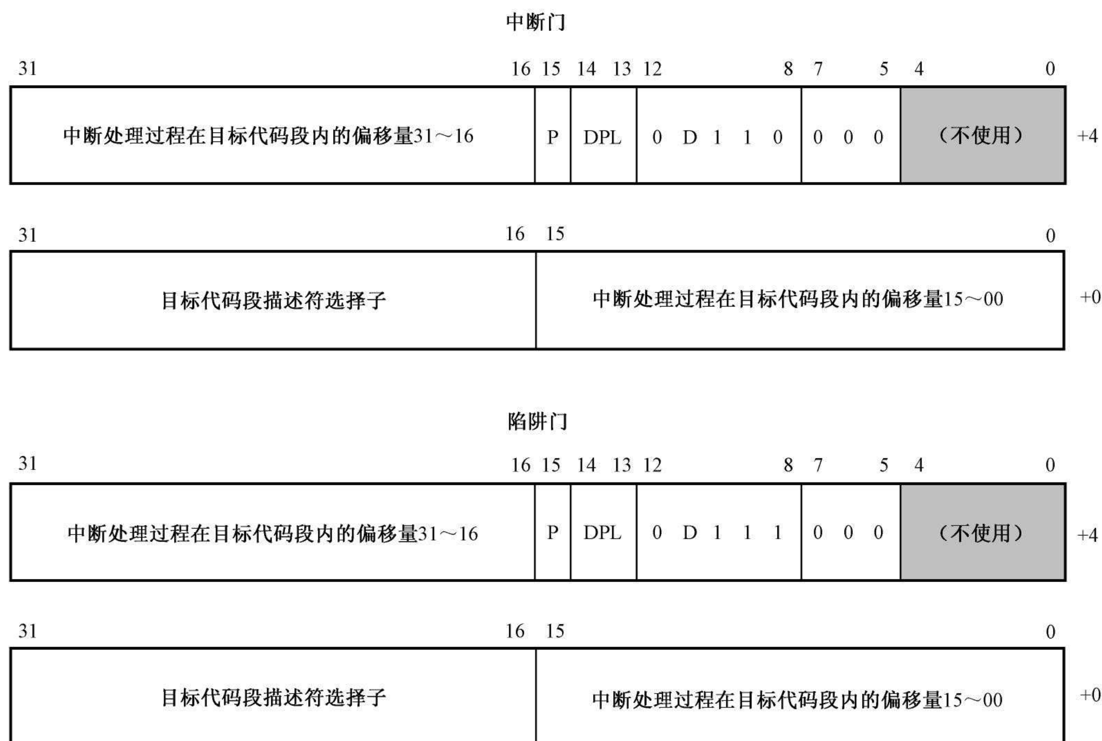


图17-1　中断门和陷阱门描述符的格式



图17-2　中断描述符表寄存器

这就意味着，中断描述符表IDT 可以位于内存中的任何地方，只要IDTR 指向了它，整个中断系统就可以正常工作。为了利用高速缓存使处理器的工作性能最大化，建议IDT 的基地址是8字节对齐的（地址的数值能够被8 整除）。处理器复位时，IDTR 的基地址部分为0，界限部分的值为0xFFFF。16 位的表界限值意味着IDT 和GDT、LDT 一样，表的大小可以是64KB，但是，事实上，因为处理器只能识别256 种中断，故通常只使用2KB，其他空余的槽位应当将描述符的P 位清零。最后，与GDT 不同的是，IDT 中的第一个描述符也是有效的。

如图17-3 所示，在保护模式下，当中断和异常发生时，处理器用中断向量乘以8 的结果去访问IDT，从中取得对应的描述符。因为IDT 在内存中的位置是由IDTR 指示的，所以这很容易做到。

注意，从图17-3 中可以看出，这里没有考虑分页，也没有考虑门描述符是任务门的情况，因为任务门的处理比较特殊。中断门和陷阱门中有目标代码段描述符的选择子，以及段内偏移量。取决于选择子的TI位，处理器访问GDT 或者LDT，取出目标代码段的描述符。接着，从目标代码段的描述符中取得目标代码段所在的基地址，再同门描述符中的偏移量相加，就得到了中断处理程序的32 位线性地址。如果没有开启分页功能，该线性地址就是物理地址；否则，送页部件转换成物理地址。注意，当处理器用中断向量访问IDT 时，要访问的位置超出了IDT 的界限，则产生常规保护异常（#GP）。

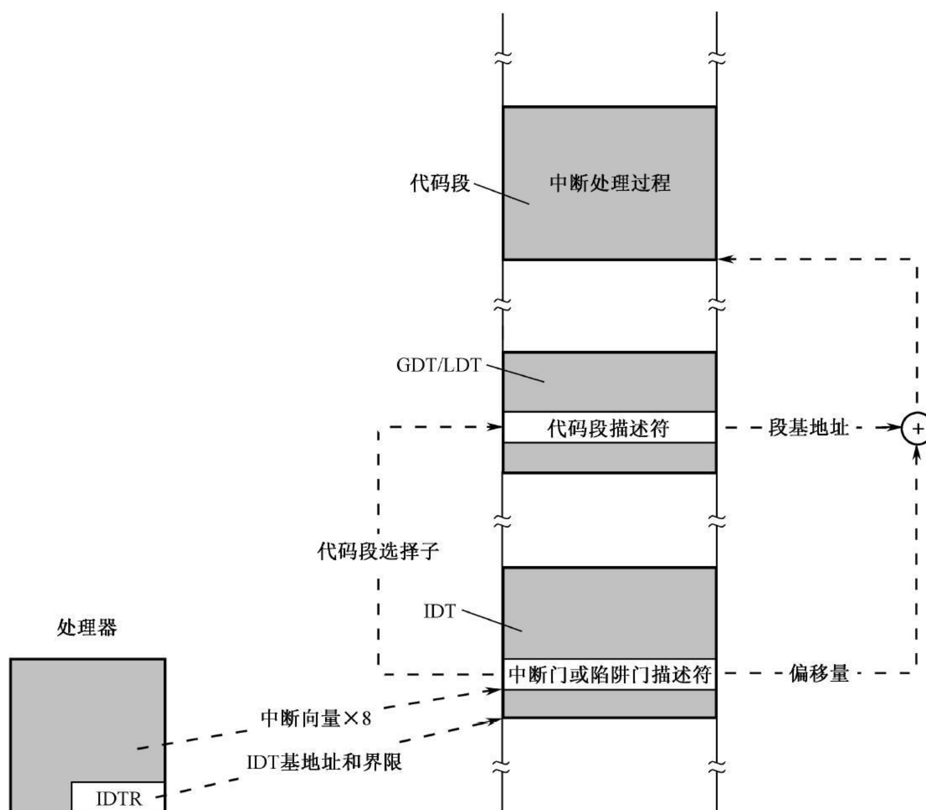


图17-3　保护模式下的中断处理过程示意图

17.1.3　中断和异常处理程序的保护

和通过调用门实施的控制转移一样，处理器要对中断和异常处理程序进行特权级保护。当目标代码段描述符的特权级（可以用门描述符中的段选择子，从GDT 或LDT 中找到）低于当前特权级CPL 时，即，在数值上，

```
CPL＜目标代码段的 DPL
```

时，不允许将控制转移到中断或异常处理程序，违反此规则将引发常规保护异常（#GP）。

不过，中断和异常处理程序的特权级保护也有一些特别之外。具体表现在：

●因为中断和异常的向量中没有RPL 字段，故当处理器进入中断或异常处理程序，或者通过任务门发起任务切换时，不检查RPL。

●中断门、陷阱门也有自己的描述符特权级DPL，即门的DPL，参见图17-1。但是，通常情况下不针对该DPL 进行检查，除了用软中断int n 和单步中断int3，以及into 引发的中断和异常。在这种情况下，当前特权级CPL 必须高于，或者和门的特权级DPL 相同，即，在数值上，

```
CPL≤门描述符的 DPL
```

这主要是为了防止低特权级的软件通过软中断指令访问一些只为内核服务的例程，如页故障处理。相反地，对于硬件中断和处理器检测到异常情况而引发的中断处理，不检查门的DPL。

中断和异常是随机产生的，不可预测。但是，有一点是可以确定的，即，它总是发生在某个任务内，是在某个任务正在执行的时候产生的，即使整个系统内只有一个任务。

当中断和异常发生时，任务可能正在特权级别为0 的全局空间（内核）中执行，也可能正在特权级别为3 的局部空间内执行。因此，当处理器将控制转移到中断或异常处理程序时，如果处理程序运行在较高的特权级别上（数值上较低的），那么，将转换栈：

●根据处理程序的特权级别，从当前任务的TSS 中取得栈段选择子和栈指针。处理器把旧栈的选择子和栈指针压入新栈。毕竟，中断处理程序也是当前任务的一部分。

●处理器把EFLAGS、CS 和EIP 的当前状态压入新栈。

●对于有错误代码的异常，处理器还要把错误代码压入新栈，紧挨着EIP 之后，如图17-4（a）所示。

●如果中断处理程序的特权级别和当前特权级别一致，则不用转换栈。

●处理器把EFLAGS、CS 和EIP 的当前状态压入当前栈。

●对于有错误代码的异常，处理器还要把错误代码压入当前栈，紧挨着EIP 之后，如图17-4（b）所示。

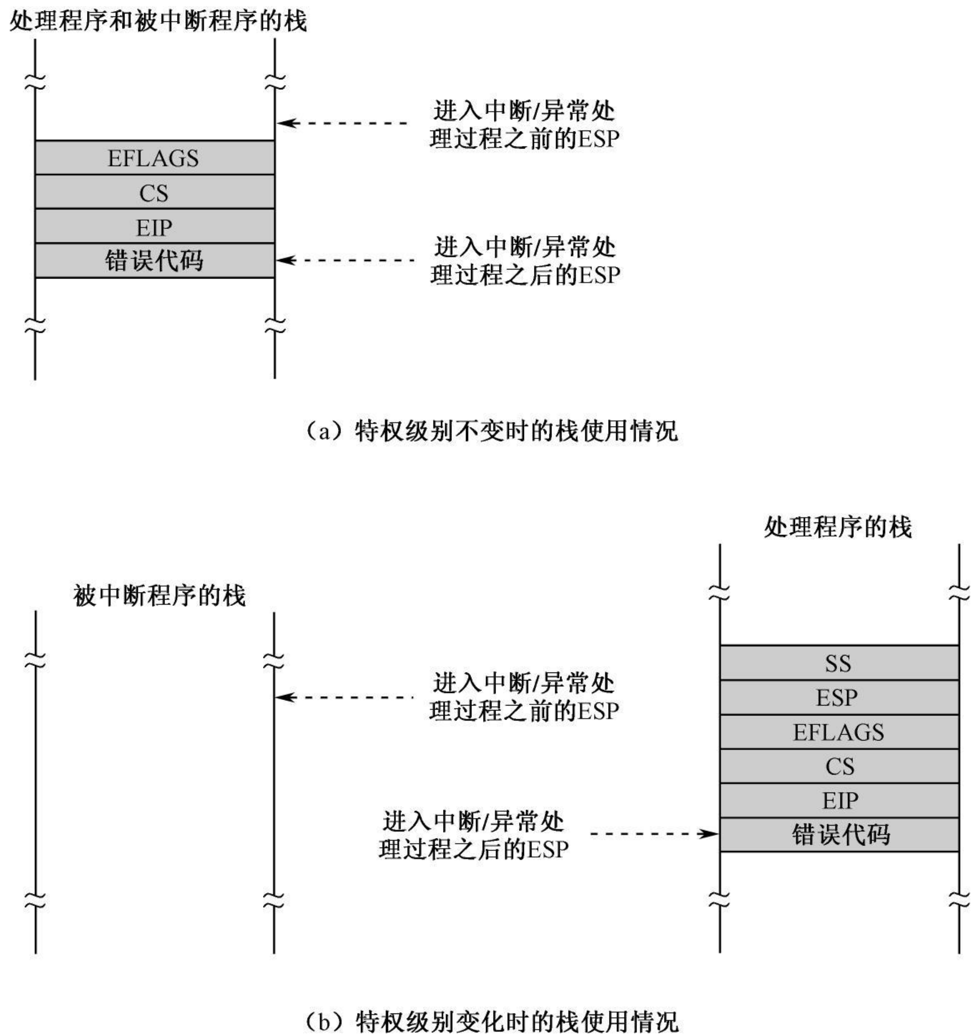


图17-4　控制转移到中断/异常处理程序时的两种栈使用情况

中断门和陷阱门的区别不大，通过中断门进入中断处理程序时，EFLAGS 寄存器的IF 位被处理器自动清零，以禁止嵌套的中断，当中断

返回时，将从栈中恢复EFLAGS 寄存器的原始状态。陷阱中断的优先级较低，当通过陷阱门进入中断处理程序时，EFLAGS 寄存器的IF 位不变，以允许其他中断优先处理。

EFLAGS 寄存器的IF 位仅影响硬件中断，对NMI、异常和int n 形式的软件中断不起作用。

17.1.4　中断任务

当中断和异常发生时，如果根据中断向量从IDT 中找到的描述符是任务门，则不是进行一般的中断处理过程，而是发起任务切换。如图17-5 所示，这是通过中断发起任务切换的原理。

用中断发起任务切换，直觉上的好处是方便。比如，因为硬件中断的发生是客观的，很容易用它来实现一个剥夺式的、抢占式的多任务系统（硬件调度机制）。

不过，这并不是它最主要的目的。想象一下，当前任务正在执行的时候，突然发生了终止类型的异常，比如双重故障（#DF），会怎么样。在这种情况下，要想用iretd 指令返回到那个任务继续正常执行已无可能。在这种情况下，如果把双重故障的处理程序定义成任务，会非常恰当。当双重故障发生时，执行任务切换，切换到内核任务中去，从容地把发生故障的任务从系统中抹去，回收内存空间，并重新调度其他任务的执行，会是最好的解决办法。具体地说，在中断机制中使用任务门可以获得以下好处：

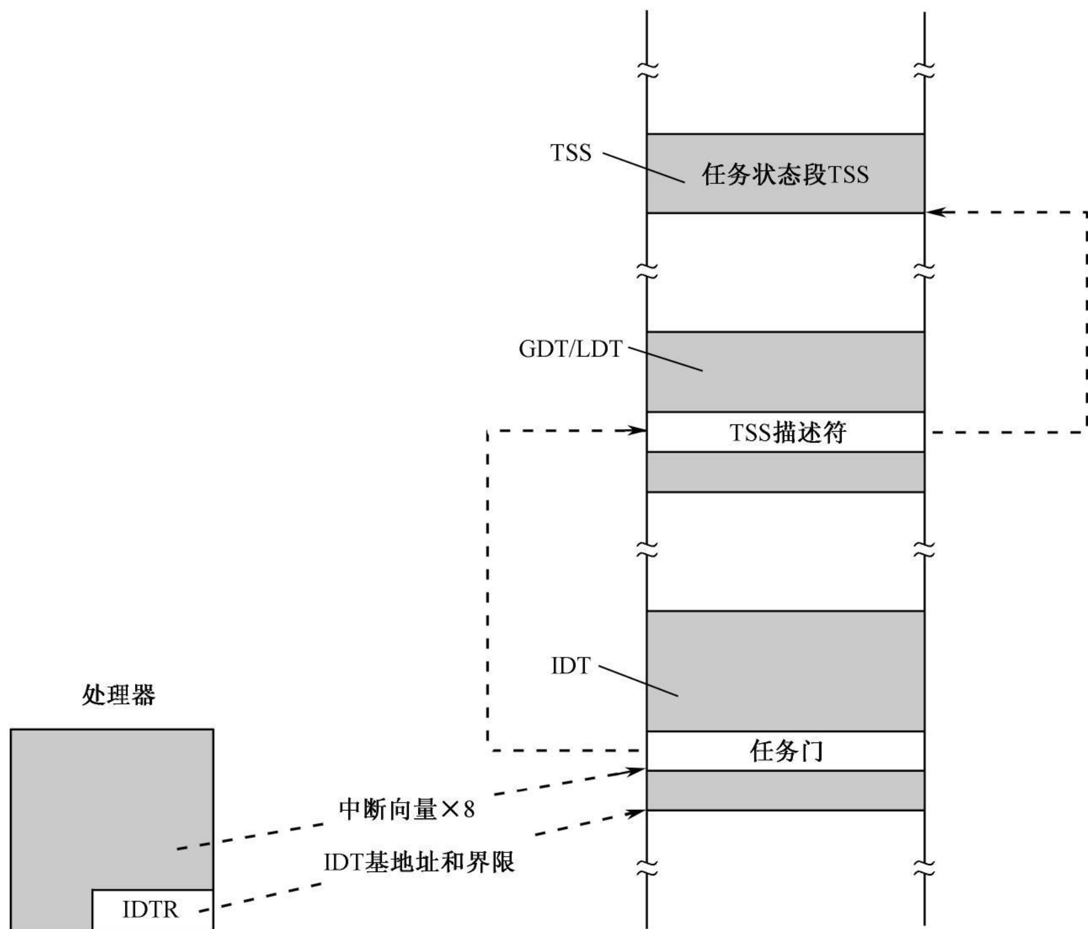


图17-5　通过中断发起的任务切换

●被中断的那个程序或任务的整个执行环境可以被完整地保存起来（保存到它的TSS 中）。

●由于接管控制的是一个新的任务，因此，可以使用一个全新的0 特权级栈。这可以有效地防止因当前任务的0 特权级栈遭到破坏而使系统崩溃。

●由于是切换到一个新任务，因此，它有一个独立的地址空间。

当然，和一般的中断处理过程相比，利用中断发起任务切换也有不利的一面，那就是速度很慢，毕竟要保存大量的机器状态，并进行一系列特权级和内存访问的检查工作。

因中断和异常而发起任务切换时，不再保存CS、EIP 的状态，但是，在任务切换工作完成后，处理器要把错误代码压入新任务的栈中（如果有错误代码的话）。

任务是不可重入的，因此，在进入中断任务之后和执行iretd 指令之前，必须关中断，以防止因相同的中断再次发生而产生常规保护异常（#GP）。

作为对任务门的保护，和中断门、陷阱门一样，只对通过int3、int n和into 指令发起的任务切换实施特权级检查，即，只有在数值上符合以下条件，才允许通过以上指令发起任务切换：

```
CPL≤任务门的 DPL
```

在其他异常和硬件中断的情况下，不检查任务门的特权级。另外，由于是任务切换，不对目标代码段的特权级别进行检查。

17.1.5　错误代码

有些异常产生时，处理器会在异常处理程序或中断任务的栈中压入一个错误代码。通常，这意味着异常和特定的段选择子或中断向量有关。

如图17-6 所示，压入栈中的错误代码是32 位的，但高16 位不用。

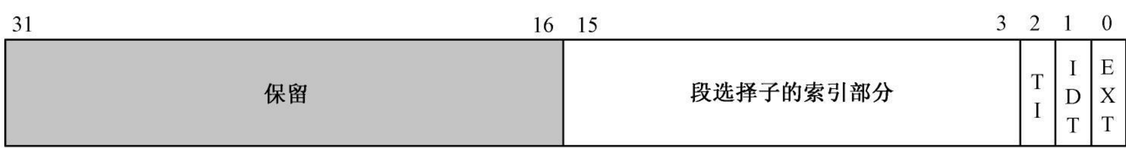


图17-6　错误代码的格式

EXT 位的意思是，异常是由外部事件引发的（External Event）。此位置位时，表示异常是由NMI、硬件中断等引发的。

IDT 位用于指示描述符的位置（Descriptor Location）。为“1”时，表示段选择子的索引部分（错误代码的位15～3）是指向中断描述符表（IDT）的；为“0”时，表示段选择子的索引部分指向GDT 或者LDT。

TI 位仅在IDT 位是“0”的情况下才有意义。此位是“0”时，表示段选择子的索引部分指向GDT，否则，指向LDT。

段选择子的索引部分用于指示GDT/LDT 内的段描述符，或者IDT 内的门描述符，它就是我们平时所用的段选择子的高13 位。

有时候，错误代码可能是全零（空），这表示异常的产生并非由于引用了一个特定的段。当然，也可能确实是在引用一个段的时候发生的，而且由于那个段的描述符是空描述符。所谓引用一个段，通常是执行了这样的指令：

```
mov ecx,0x0008
mov ds,ecx
```

注意，当通过iret/iretd 指令从中断处理程序返回时，处理器并不会自动弹出错误代码。因此，对于那些有异常代码的异常处理过程来说，必须在执行iret/iretd 指令前，先从栈中移去（或弹出）错误代码。否则，处理器在执行iret/iretd 指令时，加载（弹出）到CS 和EIP 中的返回地址就是错的。

对于外部异常（通过处理器引脚触发），以及用软中断指令int n 引发的异常，处理器不会压入错误代码，即使它原本是一个有错误代码的异常。分配给外部中断的向量号在31～255 之间，出于特殊的目的，外部的8259A 或者I/O APIC 芯片可能会给出一个0～19 的向量号，比如13（常规保护异常#GP），并希望进行异常处理。在这种情况下，处理器并不会像通常那样压入错误代码。同样地，用软中断指令

```
int 0x0d          ;向量号 13，常规保护异常#GP
```

有意引发的异常，也不会压入错误代码。

17.2 本章代码清单

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：17-1（主引导程序），源程序文件：c17_mbr.asm

本章代码清单：17-2（保护模式微型核心程序），源程序文件：c17_core.asm

本章代码清单：17-3（动态加载的用户程序/任务一），源程序文件：c17_1.asm

本章代码清单：17-4（动态加载的用户程序/任务二），源程序文件：c17_2.asm

17.3　内核的加载和初始化

17.3.1　彻底终结多段模型

平坦模型下也不是没有段，只是所有的段都很大，大到等于处理器所能寻址的全部空间。因此，在平坦模型下，至少要创建两个段描述符，一个是代码段，另一个是数据段，都是4GB。

别忘了，主引导程序的加载位置是物理地址0x00007C00，进入保护模式之后，因为不再使用多段模型，所以，只能在平坦模型下使用基地址是0x00000000 的代码段，为了继续执行程序，指令指针寄存器EIP 的初始内容必须是0x00007C00，并在此基础上随着指令的执行而增加。

现在来看代码清单17-1 的第10 行，为了使程序在平坦模型下方便地引用内存地址，这里定义了段mbr，并要求段的虚拟地址从0x00007C00开始：

```
SECTION  mbr  vstart=0x00007c00
```

如果没有vstart 子句，所有标号的地址都以程序开头为基准，从0 开始计算；一旦加了该子句，当引用一个标号时，标号所代表的地址就以程序开头为基准，从给定的虚拟地址开始计算。

言归正传，首先来创建进入保护模式时用到的段描述符，这就需要给出GDT 的物理起始地址。

第199、200 行声明了标号pgdt，并初始化了6 字节，分别是GDT 的界限值和物理地址。为了在实模式下准备全局描述符表（GDT），第12～23 行，从标号pgdt 处取得GDT 的物理地址，并计算它在实模式下的段地址和偏移地址。

在第13 章里，GDT 的物理地址是0x00007E00，现在是0x00008000，这正好是一个页的起始地址。这一变化和程序的运行无关，我只是觉得，将GDT 安排在页的开头位置比较好，仅此而已。

注意，第17 行，从标号pgdt 处取得GDT 的物理地址时，使用了指令

```
        mov eax,[cs:pgdt+0x02]            ;GDT的32位物理地址
```

当这条指令执行时，处理器工作在实模式下，段寄存器CS 的内容为0x0000。如果没有那个vstart 子句，pgdt 的地址是0x7C00+pgdt。但是，因为有vstart 子句，所以，标号pgdt 的地址是从0x00007C00 开始计算的，不用再加上0x7C00。

关于这段代码，没有什么好说的，很容易理解，已经在第13 章里讲解过了，不再赘述。

第27～32 行，在GDT 中安装最基本的两个段描述符，一个是4GB 的代码段，另一个是4GB的数据段。段的基地址都是0x00000000，段界限也都是0xFFFFF，粒度为4KB。当然，它们的属性不同。注意，没有为主引导程序创建单独的段描述符，稍后你就会看到，这实际上也不需要。

第35～47 行，为进入保护模式做准备。首先是加载全局描述符表寄存器（GDTR）；然后，打开第21 根地址线A20；最后，设置控制寄存器CR0 的PE 位，进入保护模式。

进入保护模式后，按要求，要执行一条跳转指令，以清空流水线。第50 行：

```
        jmp dword 0x0008:flush
```

其中，标号flush 是下一条指令在目标代码段内的32 位偏移地址。在第13 章里，段选择子是0x0010，所指示的段是“特制”的主引导程序段，基地址为0x00007C00，段界限是0x1FF。在那时，flush 所代表的偏移地址是相对于该段，从0 开始计算的，下一条指令的物理地址是0x00007C00+flush。

和第13 章相比，在这里，目标代码段是4GB 的段，基地址为0x00000000。由于使用了vstart子句，flush 所代表的偏移地址是从0x00007C00 开始计算的，下一条指令的物理地址是0x00000000+flush。

但是，实际上，在两种执行环境下，得到的结果是一样的。

第54～60 行，令段寄存器DS、ES、FS、GS 和SS 都指向4GB 数据段。只不过栈段是向下增长的，其他各段都向上增长。这样做，最直

接的结果就是，从此再也看不到这样的指令了：

```
mov eax,0x0008
mov es,eax
```

原因是，因为所有段都是4GB 的，用哪一个都无所谓。但是，这也带来了一个最大的好处，那就是不用在段之间换来换去，也不必记住所操作的数据位于哪个段。当它们都位于同一个4GB段时，很清爽。

第60 行的指令定义了保护模式下的初始栈，给出了栈顶所在的位置，显然，该栈是从地址0x00007000 开始向低地址方向扩展的。如图17-7 所示，GDT 位于物理地址0x00008000 处；这里定义的初始栈从物理地址0x00007000 开始向下推进，理论上，该地址以下的空间都可用于栈操作。另外还可以看出，由于当前正在执行的主引导程序并不在页的自然边界上，故，在它和GDT、栈之间出现了间隙（内存空洞）。

第63～89 行，从硬盘上加载系统内核，是从内存物理地址0x00040000 开始加载的。这段代码和第14 章一样，没有变化。

内核拥有自己独立的2GB 内存空间，而且还要被映射到每个任务的高2GB，也就是从地址0x80000000 开始的地方。为了完成这种映射，最简单的做法就是在内核代码中做点手脚，让所有对内存地址的引用（主要是对标号的引用）都从0x80000000 开始。

请看代码清单17-2，这是修改之后的内核代码。对整个代码清单稍做浏览，你会发现这里只有一个段，数据和代码混合在一起，都在同一个段内，第25 行是定义这个段的语句：

```
SECTION  core  vstart=0x80040000
```

除此之外，不再有其他段定义的语句。你可能会问，内核的虚拟地址不是0x80000000 吗？怎么会是一个奇怪的地址0x80040000 呢？

能发现这个问题的人，一定是个好学爱问的人。原因很简单，在前一章里，内核工作在多段模型下，内核代码和内核数据的重定位依赖于段的基地址。因此，在多段模型下，不管内核加载到内存中的任何位置都能正常运行。到了本章，内核工作在平坦模型下，不可能再依赖段基地址实施重定位。这怎么办呢？为了使它能够正常工作，需要用vstart 子句向编译器声明它的虚拟地址。如此一来，编译器就知道如何处理对标

号地址的引用。不过，vstart 子句中给出的虚拟地址必须和程序在运行时的虚拟地址相同。

如图17-8 所示，内核占据着物理内存的低端1MB，其核心部分的加载地址是0x00040000。在开启页功能之后，内核需要将自己映射到从虚拟地址0x80000000 开始的高端。这是一个完整的映射，很自然地，内核在高端的虚拟地址就是0x80040000 了。

一旦知道了内核运行时的虚拟地址，那么，内核代码在编译时，vstart 子句的虚拟地址也必须与之相同。只有这样，内核才能正常执行。

当然，我们知道，当内核运行时，不管是执行内核中的指令，还是访问内核中的数据，段部件所发出的线性地址都会高于0x80040000，正好位于每个任务虚拟地址空间的高端；而经页部件转换之后，得到的物理地址又变为原始的真实地址0x00040000。

再回到代码清单17-1，毕竟主引导程序还在执行呢。

第95～105 行，创建内核的页目录表和页表，并初始化必要的目录项和页表项。如图17-7 所示，页目录表的物理地址是0x00020000，第98 行，先令最后一个页目录项指向页目录表自己，即，该项所对应的页表就是当前页目录表，这是为修改页目录表而设的。接着，在页目录表内创建两个目录项，分别对应着两个不同的起始线性地址0x00000000 和0x80000000。但是实际上，它们都指向同一个页表。其中，前一个目录项只在开启页功能的时候使用，作为临时过渡。

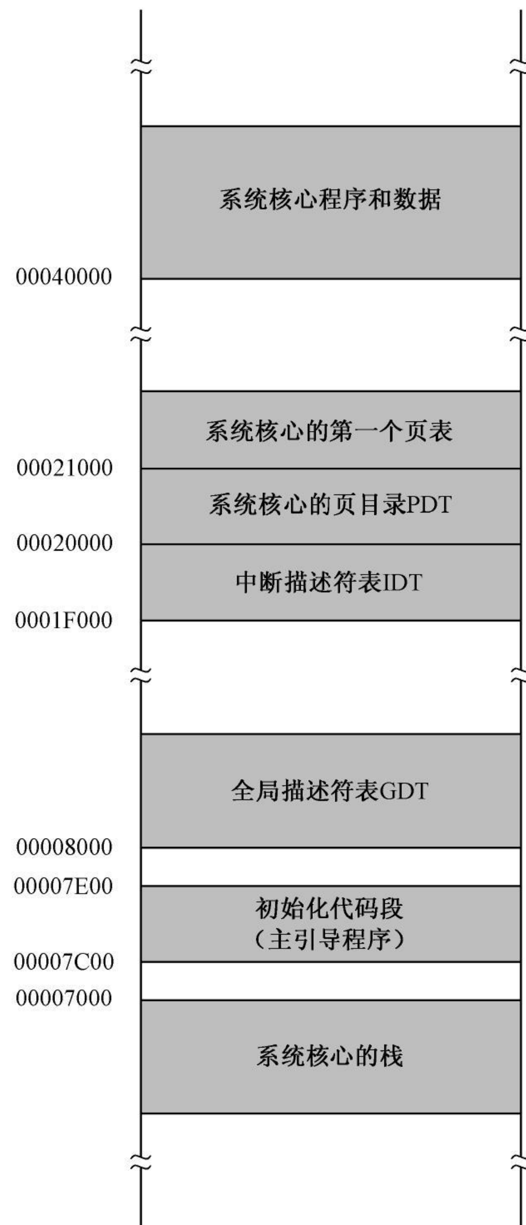


图17-7　平坦模型下的内核物理布局

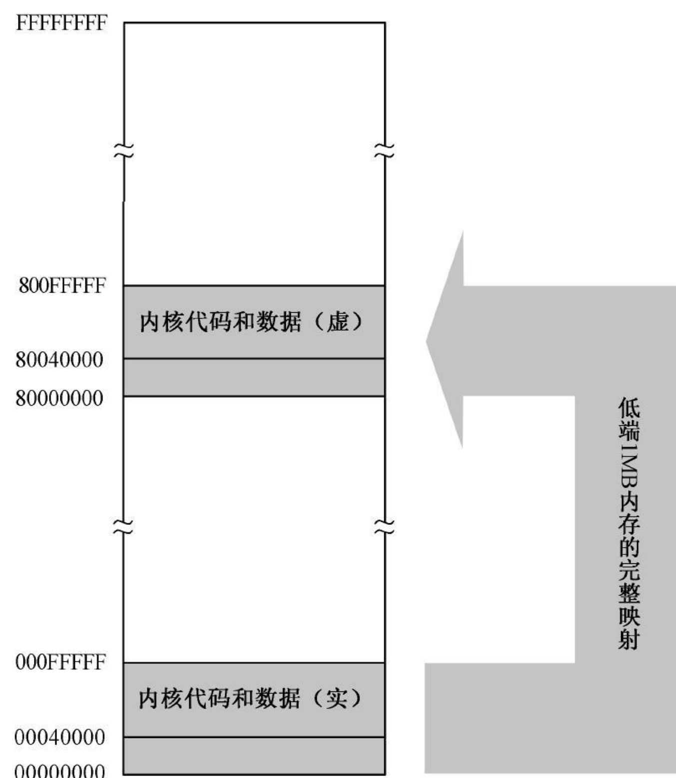


图17-8　内核区域从低端到高端的映射

页表的物理地址是0x00021000，它的前256 个页表项必须一一对应于物理内存最低端的256 个页，这是内核能正常工作的基本要求。第108～118 行，使用循环来建立这种一对一的映射关系。

第121、122 行，将内核页目录表的物理地址传送到控制寄存器CR3，这是在开启页功能之前必须要做的事情。

第125～128 行，将全局描述符表（GDT）映射到虚拟内存的高端。这也是一一映射，GDT的新地址应当是线性地址0x80000000 加上它原先的地址。

现在，内核已经从硬盘上加载完毕，页目录表和页表也已经创建。看样子一切都准备好了，第130～132 行，开启分页功能。

现在已经工作在分页模式下，和从前不同的是，不需要重新加载段寄存器CS、SS、DS、ES、FS 和GS 以刷新它们的描述符高速缓存器，因为所有这些段都是4GB 的。

关于在分页模式下工作，所有该做的工作都做了，但还是忽略了一个问题，那就是内核栈，应当将它映射到虚拟内存的高端。第137 行，通过把栈指针寄存器ESP 的内容在原来的基础上增加0x80000000，来做到这一点。

第139 行，将控制转移到内核。注意，这是一个32 位段内转移，而不是远转移（段间转移），在指令中没有使用关键字“far”。远转移在段间进行，需要16 位的目标代码段选择子，以及32 位段偏移量。在平坦模型下，所有东西都在一个大的4GB 段内，从一个地方转移到另一个位置去执行，自然是段内转移了。

事实上，这条指令有两个功能，一是转移到内核去执行，二是将处理器的执行流转移到虚拟内存的高端。内核已经从硬盘上加载，加载的位置是线性地址0x80040000。内核程序有一个头部，记载了内核的大小和入口点。参见代码清单17-2 的第25～30 行，内核程序内，偏移为0x00000004 的地方，记载着内核要执行的第一条指令的偏移量，但没有段选择子。

因此，当这条JMP 指令执行时，处理器要先访问当前代码段，从线性地址0x80040004 处取得一个32 位的段内偏移量，传送到EIP 寄存器。说时迟，那时快，内核就开始执行了。

17.3.2　创建中断描述符表

现在转到代码清单17-2，这是内核程序的代码。

当内核开始执行时，执行点位于第872 行。此时，指令指针寄存器EIP 的内容必然大于0x80040000，因为内核程序已经被映射到虚拟地址空间的高端。

接下来的工作是准备保护模式下的中断系统。保护模式下的中断机制不同于实模式，因此，在进入保护模式之前，我们已经用CLI 指令关掉了外部硬件中断，以免出现错误。而且，只有在创建了中断描述符表，并安装了中断处理程序之后，能才使用STI 指令开放硬件中断，并享受中断的好处。

如图17-7 所示，我们把中断描述符表（IDT）定义在物理内存中从地址0x0001F000 开始的地方，这里紧挨着内核的页目录表，是一段没有用到的空间。要知道，目前是在分页模式下，低端1MB 内存已经被映射

到高端，因此，中断描述符表的线性起始地址实际上是0x8001F000。该地址将多次在程序中引用，为了方便修改，在代码清单的第9 行，已经将它声明为一个常数idt_linear_address，以后可以直接将它作为数值使用。

常数定义仅仅在程序编译期间有用，在编译之后不占用任何地址空间。

表的线性地址已经确定，现在的工作是在其中安装门描述符。在这里，为每一个中断向量都定义独立的处理程序不太现实，最好是将它们归归类，比如将硬件中断归为一类，再将异常归为另一类，如此一来，只需要定义两个通用的中断处理程序即可。如果有某个中断或异常需要特殊处理，可以根据需要随时安装单独的程序。

异常的通用处理程序是在标号general_exception_handler 处定义的，位于第422 行，它只做两件事，先显示错误信息，然后停机。在屏幕上显示信息依然要使用过程put_string，在平坦模型下，调用该过程不需要使用远转移指令。但是，该过程还要被包装成调用门，以方便在用户任务内调用。通过调用门的控制转移属于远过程调用，因此，请看第59 行，put_string 过程是用retf 返回的。

这就是说，尽管过程put_string 是内核的家人，但还必须用远过程调用的方式使用：

```
mov ebx,excep_msg
call flat_4gb_code_seg_sel:put_string
```

我们只为内核定义了两个段：4GB 的代码段和4GB 的数据段，为了方便引用，在代码清单17-2 的第7 行和第8 行，分别定义了两个常数flat_4gb_code_seg_sel 和flat_4gb_data_seg_sel，前者是4GB 代码段的选择子，后者是4GB 数据段的选择子。

对异常的处理很复杂，要分具体情况。有的异常发生后，只要纠正了错误，还可以再次执行产生异常的指令，比如页故障；有的异常发生后，当前任务不可能再恢复执行；有的异常有错误代码压栈，而有的则没有。前两种情况还好办，如果你愿意，还能用iretd 指令返回；但是对于有和没有错误代码的情况，就不好办了。没有还好，可以直接用iretd 返回；如果有，则必须先弹出错误代码。在一个通用的异常处理程序

中，无法判断有没有错误代码压栈，因此，唯一的异常处理办法就是停机。

第877～880 行，调用make_gate_descriptor 过程创建中断门描述符。该过程不但可以用于创建调用门描述符，还可以用来创建中断门、陷阱门和任务门的描述符。在此处，描述符的目标代码段选择子是当前内核4GB 代码段选择子；段内偏移量是通用异常处理过程的线性地址，肯定大于0x80040000；描述符的属性值为0x8E00，指示这是一个32 位的中断门描述符，门的特权级别为零。

一般来说，内核不会允许3 特权级的用户任务使用make_gate_descriptor 过程。因此，它可以定义成用ret 指令返回的近过程，而不是现在的远过程。在内核中以近过程调用的方式使用它更方便，但我们也不想对它做任何改动，毕竟它一直是用retf 指令返回的。因此，在这里对它的调用还是远调用。

第882～889 行，在IDT 中安装前20 个描述符，它们都指向通用异常处理程序。EBX 寄存器指向IDT 的线性基地址；ESI 寄存器是IDT 内的索引，或者说是中断向量号。每个描述符占8 字节，因此，每个描述符的线性地址是EBX＋ESI×8。

第892～903 行，在IDT 内安装通用中断处理程序，中断向量20～255，对应着Intel 保留的中断向量，以及外部硬件中断。通用的中断处理过程general_interrupt_handler 是在第410 行定义的。第411～419 行，先是向8259A 芯片发送中断结束命令EOI（End Of Interrupt），然后执行iretd 指令从中断返回。很明显，通用的中断处理过程什么也不做。但是，如果没有这个什么也不做的过程，当中断发生时，就会出问题。

根据实际需要，中断或异常应当单独处理。第906～913 行，在IDT 中安装0x70 号中断的处理过程。这段代码很好理解，首先用make_gate_descriptor 过程创建一个指向0x70 号中断处理过程的中断门描述符，然后，将它写入相应的IDT 表项内。该表项的线性地址等于IDT 的线性起始地址，加上0x70 乘以8。

17.3.3　用定时中断实施任务切换

刚才安装的那个0x70 号中断处理过程，主要目的是进行任务切换。

我们知道，计算机主板上有实时时钟芯片RTC，可以定时产生更新周期结束中断信号。可以设置RTC 芯片，使得它每次更新CMOS 中的时间信息后，便发出这个中断信号。在本书的前半部分，刚开始引入中断的概念时，我们用过这个中断。

RTC 芯片的中断线和8259A 从片的第1 个引脚相连，一般情况下，该引脚对应的中断向量为0x70。因此，它的处理过程就叫rtm_0x70_interrupt_handle，位于代码清单17-2 的第429 行。

由于是硬件中断，因此，第433～435 行，先要向8259A 芯片发送中断结束命令EOI，否则它不会再向处理器发送另一个中断“通知”。

说实在的，用实时时钟的更新周期结束中断来实施任务切换并不是一个好主意。和别的中断相比，它更啰嗦，因为必须读一下CMOS 芯片内的寄存器C，使它复位一下，才能使RTC 产生下一个中断信号。否则，它只产生一次中断信号。因此，第437～439 行就用来做这个工作。如果对此不熟悉，建议回到本书的前面复习一下。

在多任务系统中，同时有很多任务等待调度。为了记住都有哪些任务，我们使用了任务控制块（TCB），并把它们穿在一起，形成TCB 链，链上的每一个TCB 称为节点。在上一章里，图16-25 给出了TCB 的基本结构。在这一章里，我们继续使用这个版本的TCB。

学过数据结构的人都知道，链表用得很广泛，而它也拥有一套完整的算法，用来添加节点、插入节点、删除节点和遍历整个链表。很荣幸地，我们现在终于有机会用汇编语言实现这些算法了。

在我们这个链表中，有一个链表头，指向第一个TCB 的线性基地址。然后，在每个TCB 内偏移量为0x00 处，是下一个TCB 的线性地址。当此处为0 时，说明这是链上最后一个TCB。第517 行，声明了标号tcb_chain，并初始化了一个双字，这就是链表头。如图17-9所示，链表头有自己的线性地址，比如0x8005320。它是一个双字，内容是0x80006000，这就是链上第一个TCB（TCB1）的线性地址。

在线性地址0x80006000 处，是TCB2，其内部偏移量为0x04 的地方，是当前任务的状态，这是一个字，若其值为0x0000，表示这是一个空闲的任务，或者一个挂起的任务；若其值为0xFFFF，则表明这是当前正在运行中的任务（当前任务，或者忙任务）。在任务时候，链表中只允许一个为忙的任务。

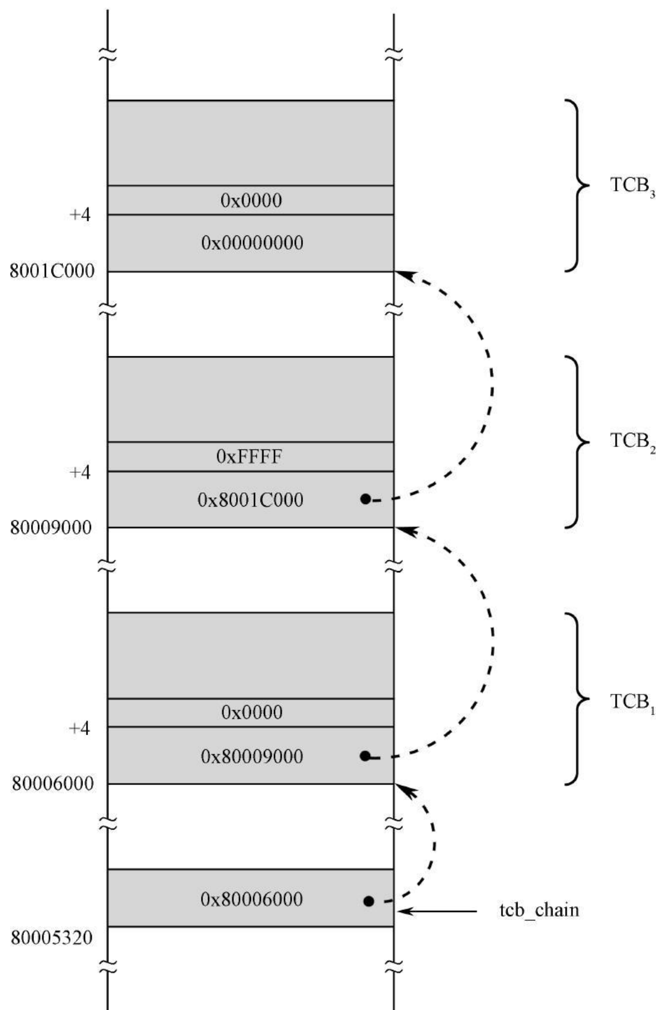


图17-9　TCB 链表示意图

TCB_1 内，偏移为0x00处，是下一个TCB，即TCB_2的线性地址。于是，我们可以根据0x80009000 这个值，定位到TCB_2。很显然，这是一个正在运行中的任务，状态为忙，下一个TCB，即TCB_3的线性地址是0x80001C00。再来看TCB3，它的状态为挂起或者空闲，而且内部偏移为0x00 的地方是0，它就是链上最后一个TCB。

在中断内实施任务切换，可以使用jmp 指令，从当前正在运行的任务切换到另一个空闲任务。中断的发生是随机的，但是，可以肯定的是，当中断发生时，必定有一个任务正在运行中。因此，中断总是在某个任务内发生的。

如图17-10 所示，当中断发生时，任务可能正在局部空间执行，也可能正在全局空间内执行，即在内核中执行，毕竟内核被映射到每个任务地址空间的高2GB。无论是在任务的局部空间执行，还是在全局空间执行，当中断发生时，因为中断处理过程位于内核中，因此，控制都会转移到任务的全局空间，去执行当前的中断处理过程rtm_0x70_interrupt_handle。

所有任务都共用同一个全局空间，因此，中断处理过程rtm_0x70_interrupt_handle 也只有一份。尽管如此，当某个任务成为正在执行的当前任务时，它便拥有了该中断处理过程。每个任务在执行该过程时都有自己独立的机器状态和寄存器状态，并使用自己私有的0 特权级栈段。所以，这里面不存在任何冲突和混乱的情况。

在图17-10 中，我们是假定中断发生在任务的局部空间。也就是说，任务正在自己的局部空间内执行。此时，将转到全局空间内执行内核的中断处理过程。

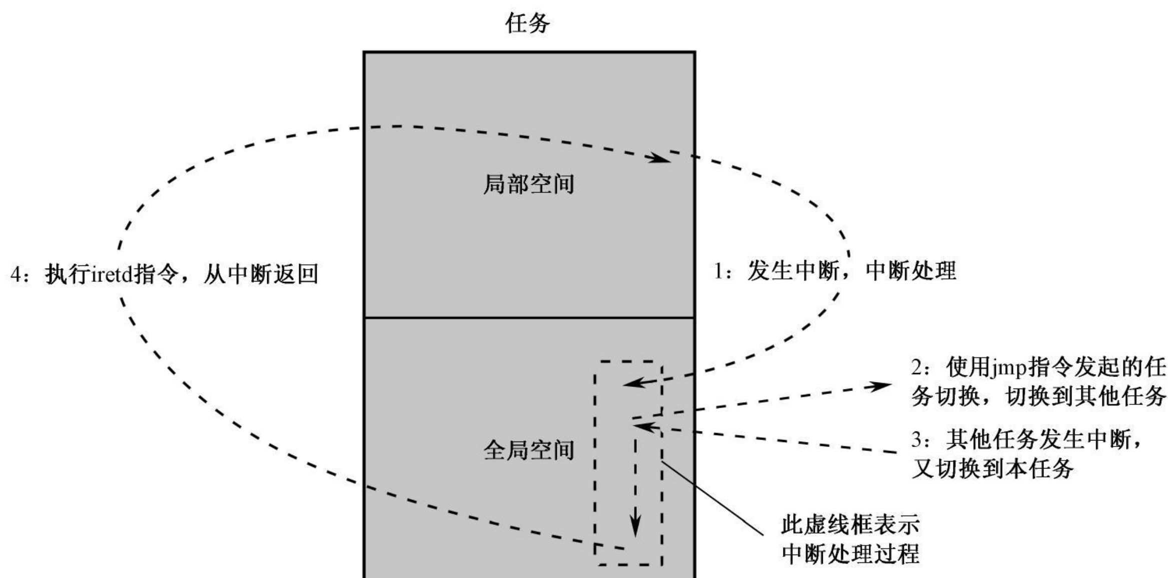


图17-10　利用硬件中断实施任务切换的全过程

中断处理过程的主要功能是确定下一个应该被执行的任务，并切换到那个任务。整个过程如下：

① 遍历TCB 链，找到当前任务，也就是寻找那个状态值为0xFFFF的节点。如果找不到，或者链表为空，则直接转到步骤⑥。

② 如果找到了，则将此节点移到链表的末端，使其成为最后一个节点。因为该任务刚刚执行完，所以，将它移到链尾，可以使其被调度的优先级别最低。

③ 再次遍历TCB 链，寻找链上第一个状态为空闲的任务，也就是寻找状态值为0x0000 的节点。如果找不到，则直接转到步骤⑥。

④ 如果找到了，将当前任务的状态置为0x0000，将找到的空闲任务的状态置为0xFFFF。

⑤ 使用jmp 指令从当前任务切换到空闲任务。

⑥ 执行iretd 指令，中断返回。

接着看图17-10，一旦找到了当前为忙的任务，以及那个空闲任务，则按图中所示，使用jmp指令发起任务切换，切换到空闲任务。因为用的

是jmp 指令，故当前任务的TSS 描述符的B 位变成“0”，而新任务TSS 描述符的B 位变成“1”，当前任务和新任务之间是非嵌套的。

另外，非常明显的是，当中断发生，控制转移到其他任务的时候，当前（旧）任务的状态是停留在中断处理过程中的，该任务的TSS 可以保存这一状态。当下一次从其他任务切换到这个任务后，将继续执行未完成的中断处理过程，并在过程的最后执行iretd 指令，于是返回到当初发生中断的地方继续执行。在图17-10 中，是返回到任务的局部空间执行。

注意，**其他任务的执行情况也和图17-10 中的这个任务相同。**

一旦明白了我们要做什么，以及如何做，现在，来看看这个过程具体是怎么实现的。首先是在链表中找到当前任务，也就是那个状态为忙（0xFFFF）的节点（Node），这是代码清单17-2 第 442～450 行的功能。

第442 行，先把链表头tcb_chain 的线性地址传送到EAX 寄存器；第444 行，因为链表头的内容是第一个TCB 的线性地址，因此，EBX 寄存器的内容就是第一个TCB 的线性地址，如图17-11（a）所示。

第445、446 行，判断EBX 寄存器的内容是否为0。也就是说，链表是否为空。如果是仅有一个链表头的空表，就直接转移标号.irtn 处，那里是一个从中断返回的iretd 指令。否则，说明链表不空，于是EBX 寄存器的内容就是第一个TCB 的线性地址，如图17-11（b）所示。

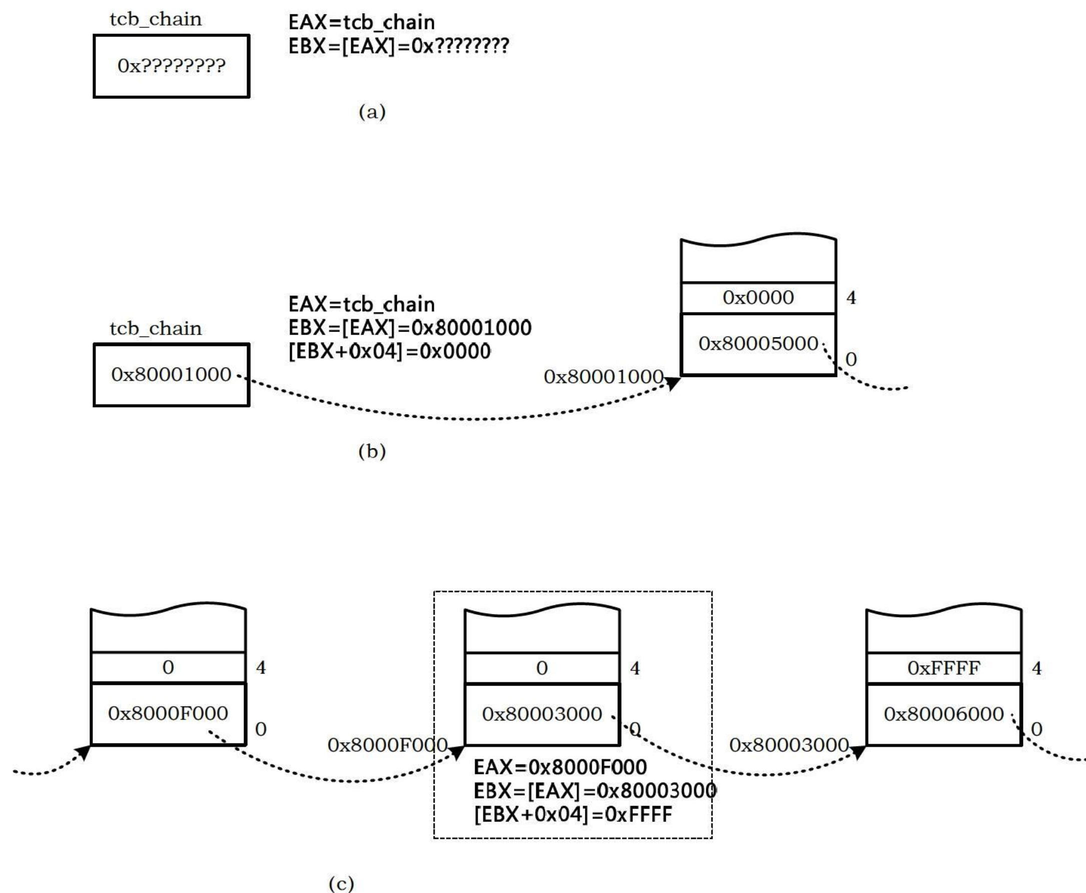


图17-11　链表遍历过程中的当前节点

第447、448 行，判断链上第一个TCB 是否为当前正在执行的任务。即，TCB 内的状态域是否为0xFFFF。如果不是，那么，将EBX 寄存器的内容传送到EAX，即，令EAX 寄存器指向第一个TCB 的线性地址，回到前面，重复执行第444～450 行的指令，顺着链表继续往后查找。

如图17-11（c）所示，无论在哪种情况下，EAX 寄存器总是指向当前节点（包括头节点tcb_chain），而EBX 寄存器总是指向下一个节点。要是能够在链表中找到一个状态值为0xFFFF（忙）的节点，那么，它必然是当前节点的下一个节点。也就是说，EAX 寄存器指向当前节点，EBX 寄存器指向那个状态值为0xFFFF 的节点。注意，这里的“当前节点”包括头节点tcb_chain 在内。

另一个值得回味的事情是，由于是在平坦模型下，链表头tcb_chain 和其他节点现在都位于同一个4GB 段内，所以才有了这段比较“流畅”、“自然”、“统一”的处理过程。要是在以前，头节点tcb_chain 位于一个较

小的内核数据段；其他节点则在可用的物理内存中，或者任务的虚拟地址空间内动态创建，由于它们不在一个段内，无法用同一段代码进行处理。

只有在找到一个状态值为0xFFFF 的节点时，处理器的执行流程才会到达标号.b1 处。此处的任务是将该节点移到链表的末端，使它能够被再次调度执行的可能性降到最低，毕竟它刚刚执行过。

为此，第454、455 行，先将那个状态为忙的节点从链表中拆除。如图17-12 所示，该节点是当前节点的下一个节点，即节点A，其线性地址在EBX 寄存器中，故第454 行可以取得该节点的下一个节点，即节点B的线性地址。然后，第455 行，把节点B 的线性地址填写到当前节点的“下一个TCB 线性地址”域中。

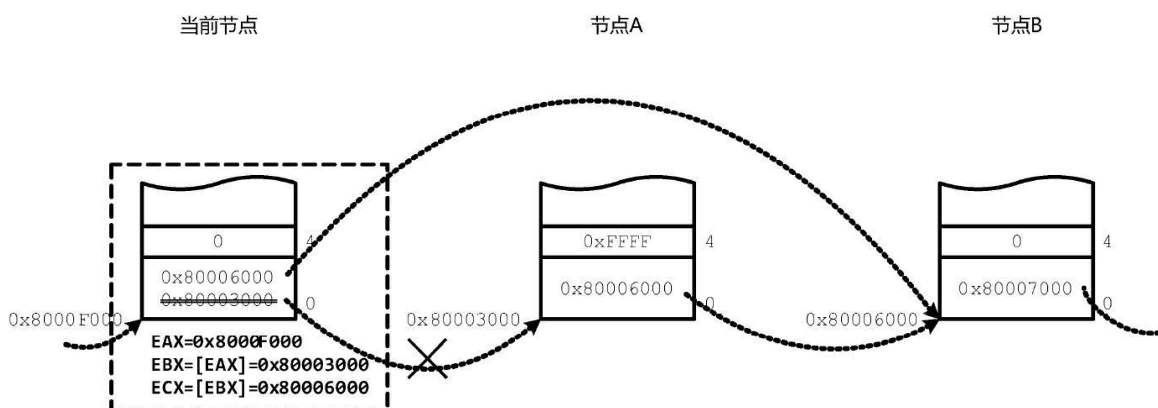


图17-12　将状态为忙的节点从链上拆除

作为一个特别的例子，如图17-13 所示，如果链上只有一个TCB，而且是个状态值为0xFFFF的节点，那么，当前的算法也同样适用。在这种情况下，链表头将失去和这个唯一的节点的联系。不过，这只是暂时的。接着往下看，我们要将那个被拆除的节点挂到链表的末端。

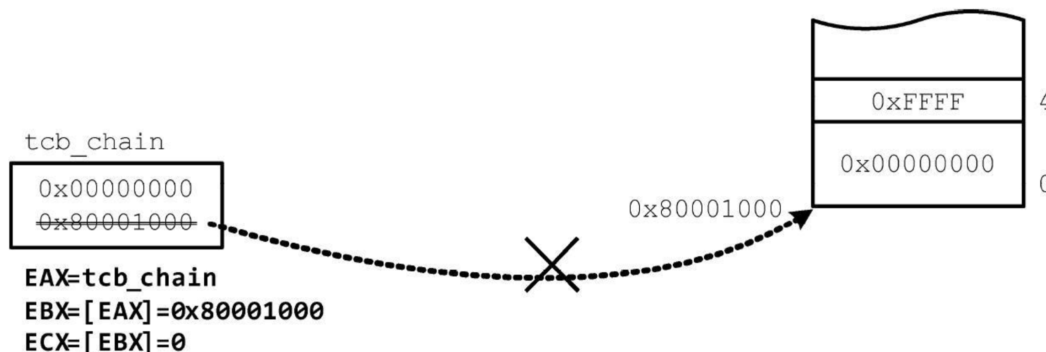


图17-13　从链上拆除节点的算法适用于只有一个TCB 的情况

第458～462 行，从当前节点开始，继续往后遍历链表，直到链表结束。当前节点的线性地址总是在EAX 寄存器中，而EDX 寄存器的内容总是下一个节点的线性地址。这段代码将反复执行，每次都把下一个节点作为当前节点来处理，就这样，最终会定位到链上最后一个节点。

如图17-14 所示，当执行流程到达标号.b3 处时，EAX 寄存器指向链上最后一个节点，其内容为该节点的线性地址；EBX 寄存器在这段代码中没有使用，故它还指向那个状态值为0xFFFF 的任务；EDX 寄存器的内容始终指向当前节点的下一个节点，遍历到链表末端时，其内容为0。

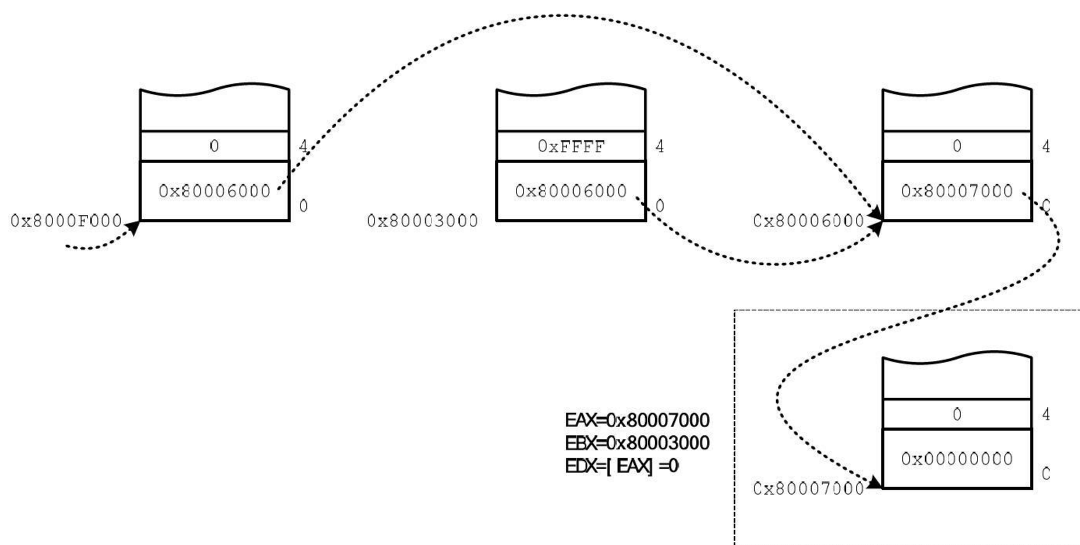


图17-14　遍历链表，直到最后一个节点

第465、466 行，将状态值为0xFFFF 的节点挂到链表的末端，同时，将它的“下一个TCB 线性地址”域改为0，表明这是最后一个节点，如图17-15 所示。

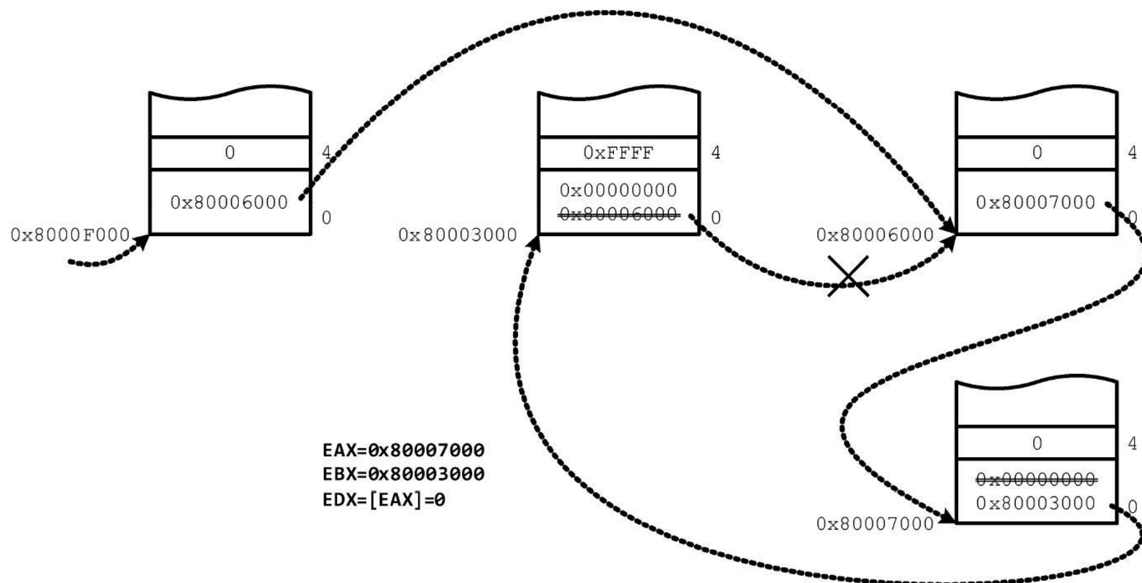


图17-15　将状态为忙的任务挂在链表的末端

对于前面那个特别的例子来说，即使链上只有一个TCB，将它从链中拆除，并移到链尾的过程也同样适用。如图17-16 所示，当那个唯一的、状态值为0xFFFF 的节点从链中拆除后，链表头tcb_chain 中的内容为0。当执行遍历动作时，它会被当成链上最后一个节点对待。在这种情况下，最终的结果是又重新将那个被拆除的节点挂在链表上，前后的结果一样，来回一样远。

第469～475 行，从头开始遍历链表，直到发现第一个状态值为0x0000（空闲）的节点。这段代码没什么好说的，方法和前面一样。如果没有找到，就转到标号.irtn 处，直接从中断返回。原因是，如果没有状态为空闲的任务，任务切换就没办法进行。这种可能性是有的，比如，在内核刚刚加载，并做为第一个状态为忙的任务添加到链表中的时候。这时，链表中只有一个TCB，而且节点的状态值为0xFFFF。

不过，一旦发现有空闲的任务，那么此时EAX 寄存器指向这个状态值为0x0000（空闲）的节点，EBX 寄存器指向那个状态值为0xFFFF（忙）的节点。第478、479 行，分别用not 指令将它们的状态值取反。也就是说，原先为空闲的任务变忙；原先为忙的任务变为空闲。

在每个TCB 内偏移量为0x14 的地方，依次是该任务TSS 的线性地址和TSS 描述符选择子。第450 行，使用间接远转移指令jmp 发起任务切换，控制转移到被选中的那个新任务。

等下一次当前任务又获得执行权时，返回点是第483 行。于是，执行iretd 指令，从中断返回，正常执行任务的其他代码，直到下一个实时时钟中断发生。

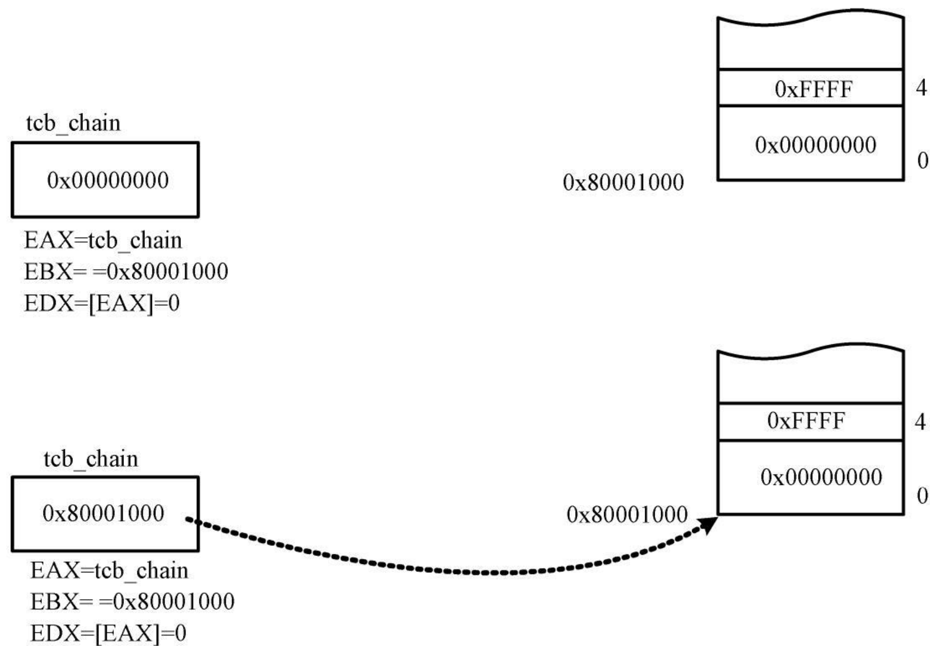


图17-16　将节点移到链尾的特别例子

17.3.4　8259A 芯片的初始化

看起来，和所有中断/异常相关的门描述符都已经安装好了，现在，应该把中断描述符表的基地址和界限值加载到中断描述符表寄存器（IDTR）中。

第916 行，将中断描述符表的界限值写入标号pidt 处的字单元。pidt 是在第513 行声明的，初始化了6 字节，其中，前面的字是IDT 的界限值，等于描述符的个数（255）乘以8 减去1；后一个双字保存的则是IDT 的线性基地址。IDT 的基地址已经定义为常量idt_linear_address，因此，第917 行，将这个数值写入该双字单元中。

第918 行，用lidt 指令加载IDTR 寄存器（Load Interrupt Descriptor Table Register）。该指令的格式和lgdt 相同：

```
lidt m48            ;lidt m16&m32
```

这就是说，该指令的操作数是一个内存地址，指向一个包含了48 位（6 字节）数据的内存区域。在16 位模式下，该地址是16 位的；在32 位模式下，该地址是32 位的。和lgdt 指令一样，该指令在实模式下也可以执行，以便于在进入保护模式之前就做好有中断有关的准备工作。

在这6 字节的内存区域中，前16 位是IDT 的界限值，高32 位是IDT 的线性基地址。在初始状态下（计算机启动之后），GDTR 的基地址被初始化为0x00000000；界限值为0xFFFF。

该指向不影响任何标志位。

一旦设置了中断描述符表（IDT），并加载了IDTR 寄存器，处理器的中断机制就开始起作用了。比如，要是有异常发生，就会调用相应的异常处理过程。如果EFLAGS 寄存器的IF 位处于置位状态，硬件中断也能得到相应的处理。不过，依目前的状态，还不宜开放硬件中断。

在保护模式下，如果计算机系统的可编程中断控制器芯片还是8259A，那就得重新进行初始化。事实上，8259A 并没有过时，在单处理器系统中，它依然健在。

重新初始化8259A 芯片的原因是其主片的中断向量和处理器的异常向量冲突。计算机启动之后，主片的中断向量为0x08～0x0F；从片的中断向量是0x70～0x77，在以8086 为处理器的系统中，这没有什么问题，在32 位处理器上，0x08～0x0F 已经被处理器用做异常向量。

好在8259A（以及I/O APIC）都是可编程的，允许重新设置中断向量。根据Intel 公司的建议，中断向量0x20～0xFF（32～255）是用户可以自由分配的部分。那么，我们可以设置8259A的主片，把它的中断向量改成0x20～0x27，这样就没问题了。

对8259A 编程需要使用初始化命令字（Initialize Command Word，ICW），以设置它的工作方式，共有4 个初始化命令字，分别是ICW1～ICW4，都是单字节命令。ICW1 用于设置中断请求的触发方式，以及级联的芯片数量；ICW2 用于设置每个芯片的中断向量；ICW3 用于指定用哪个引脚实现芯片的级联；ICW4 用于控制芯片的工作方式。

对8259A 芯片的编程不是本书的重点，因为这涉及它的内部构造和工作原理，说来话长。同时，这还是一个令人厌恶的芯片，只分配了两个端口，设置起来拐弯抹角，很麻烦。不像有些芯片，每个端口对应着一个命令字，比较简单。

主片的端口号是0x20 和0x21，从片的端口号是0xA0 和0xA1，要发送初始化命令字给8259A 芯片，对于主片来说，需要先向0x20 端口发送ICW1，而对于从片来说，这个端口是0xA0。这是一个标志，每次8259A 芯片接到ICW1 时，都意味着一个新的初始化过程开始了。

从0x20/0xA0 端口接受命令字ICW1 后，8259A 芯片期待从0x21/0xA1 端口接受命令字ICW2。但是，它是否期待ICW3 和ICW4，还要看ICW1 的内容。如图17-17 所示，ICW1 的位0决定了是否有ICW4 命令，位1 指示是否为多片级联。如果是多片级联，那么，必定有ICW3 命令。这样一来，8259A 芯片就知道，在接受了ICW2 命令之后，是否还要在相同的端口（0x21/0xA1）上依次再接受ICW3 和ICW4。

注意，在图17-17 中，深色的比特位表示它已被保留，或者不用，使用图中所标注的固定值（0 或1）；有些比特虽然不是深色，但也标注了固定值（0 或1），这些位是有意义的，可以设置或改变，具体的含义可参考芯片手册。但是，之所以在这里采用固定值，是因为就目前的应用环境来说，这是比较通用的合理设置。

来看代码清单17-2。

第921、922 行，先向8259A 主片发送ICW1，端口号是0x20。从命令上看，这里需要ICW4，而且指定了多芯片级联方式，中断信号的采集用的是边沿触发方式。因为是多芯片级联，故需要ICW3。

第923、924 行，通过另一个端口0x21 向主片发送ICW2 命令。如图17-17 所示，ICW2 命令用于设置芯片的中断向量号。芯片每个引脚的中断向量号不需要单独设置，只需要一个起始向量号即可。ICW2 的低3 位不用，固定为0，仅高5 位有效。在这里，ICW2 的值是0x20，对应着二进制数00100000，高5 位是00100。此时，该芯片的8 个中断引脚就分别对应着中断向量号0x20～0x27。

再举个例子，如果ICW2 的高5 位是01101，那么，加上低3 位的全“0”，它对应的二进制数就是01101000，即0x68，该芯片的中断向量为0x68～0x6F。

第925、926 行，依然通过端口0x21 向主片发送ICW3 命令。如图17-17 所示，发送给主片的命令和发送给从片的命令，是不相同的。因为这里是在设置主片，故该命令字的7 比特分别表示那个引脚是否连着从片。从命令字上看，是0x04，即二进制的00000100，也就是说，该芯片的第3 个引脚连着从片。

第927、928 行，依然通过端口0x21 向主片发送ICW4 命令。如图17-17 所示，我们发送的命令字是0x01，这表示要求采用非自动结束方式。对于单片使用的场合，采用自动结束方式较为方便，但多片级联的场合，应当采用非自动结束方式。

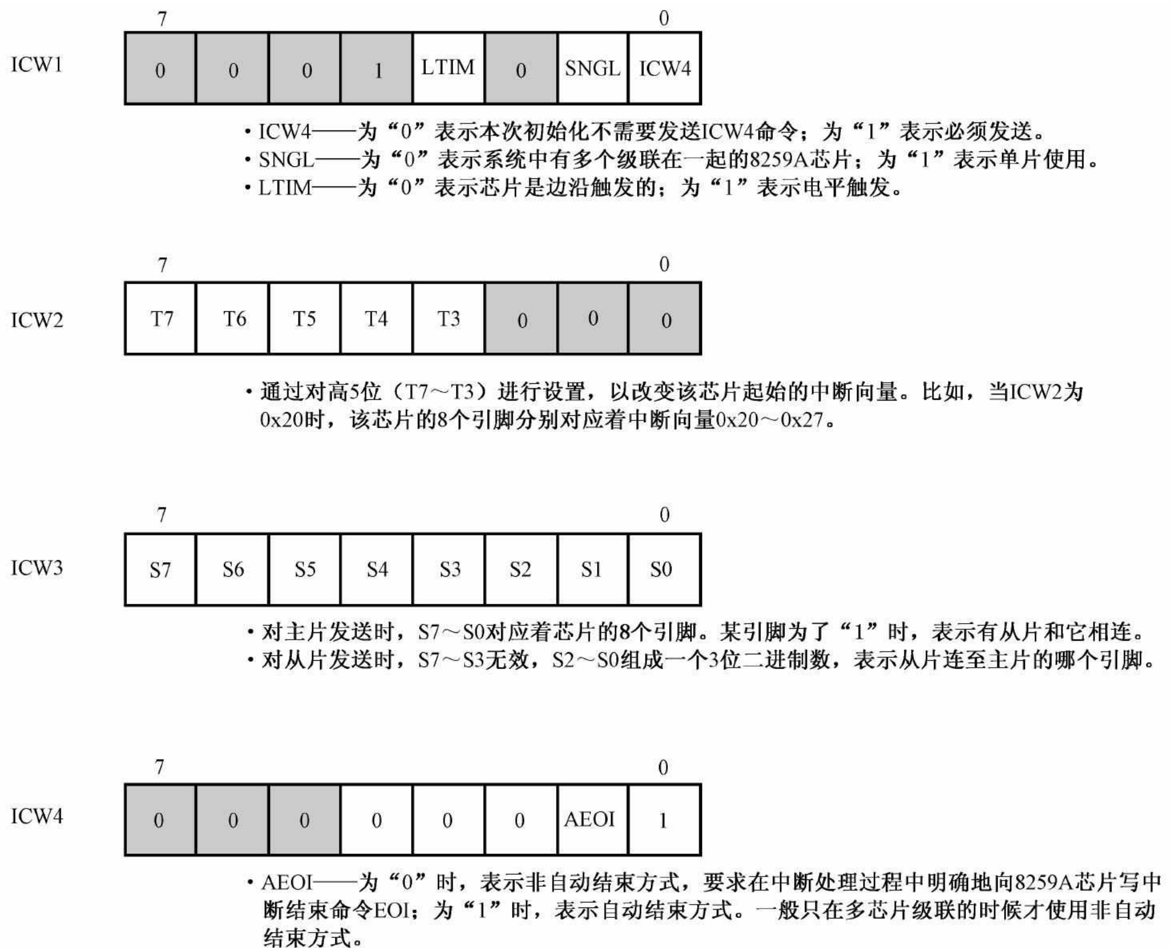


图17-17　8259A 芯片的初始化命令字

第930～937 行，这些代码用于设置和主片相连的从片，方法大致相同，读者自行分析。

第940～952 行，这段代码专门用于设置和0x70 号时钟中断有关的硬件状态，包括RTC 和8259A。对RTC 的设置包括允许它产生哪些中断信号，并读一下它的寄存器C。寄存器C 在每次读取后自动清零，如果没有清零，RTC 将不会产生中断信号；对8259A 的设置主要是打通它和RTC 之间的中断信号通路。这段代码是从第9 章原封不动地抄来的，在那一章里已经做过讲解，这里不再赘述。

第954 行，用sti 指令设置EFLAGS 寄存器的IF 位，开放硬件中断。

中断是计算机系统中一个必不可少的恶魔，不用它便罢，一旦放出了它，就好比打开了潘多拉魔盒。从此之后，如果你处理不当，各种奇怪的程序问题都有可能出现，而且神出鬼没，不容易找到它发生的根源。

这是可以理解的。在一个顺序工作的程序中，很容易用调试工具找到错误指令和出错原因。但是，中断是随机发生的，而且不能确定在中断发生时，处理器将控制转移到了哪里。即使知道出错的位置，也不容易发现错误的原因。很多时候，中断处理过程和被中断的程序有着逻辑上的关联，包括状态的依赖和数据的共享和争用，等等。

说这些就扯远了，还是看看现在都会发生什么。因为执行了sti，硬件中断会随时得到处理。特别是我们最关注的实时时钟中断，它差不多会在1 秒钟内发生一次。当此中断发生后，过程rtm_0x70_interrupt_handle 就会被执行。它会遍历TCB 链，找到一个状态为忙的任务，和一个状态为空闲的任务，然后发起任务切换。

就目前的实际情况而言，该中断处理过程不会做太多的事，仅仅是给8259A 芯片发送中断结束命令EOI，并读一下RTC 芯片的寄存器C，然后执行iretd 指令返回，因为目前链表为空。

17.3.5 平坦模型下的字符串显示例程

代码清单17-2 第956、957 行，在屏幕上显示字符串，表示内核正工作在保护和分页模式下，内核的地址空间已经被映射到地址0x80000000 以上。

在本章中，由于使用了平坦模型，put_string 过程也不得不做了大幅度修改，以适应这种变化。一直以来，该过程接受的参数是DS:EBX。此处，DS 是数据段寄存器，要显示的字符串位于它所指向的段中；EBX 寄存器的内容是字符串在段内的偏移量。显然，老版本的put_string 过程是面向多段模型的。

相反地，该过程的新版本只能工作在平坦模型下，它只需要用EBX 寄存器传入字符串的线性地址即可。在平坦模型下，字符串位于任务的4GB 虚拟地址空间内，它的线性地址是唯一的。

在字符串的显示期间，需要临时关闭硬件中断，即，用cli 指令清零EFLAGS 寄存器的IF位。如果不这么做，那么，在字符串显示期间，随时会被中断。如果切换到另外一个任务，那么，两个任务所显示的内容就有可能在屏幕上交替出现。当然，这还算不上是严重的问题，更严重的是写光标寄存器，举个例子，任务A 读光标位置，并在屏幕上写了一个字符。当它正准备用新的数值写光标寄存器时，中断发生，任务B 开始执行。任务B 也读光标位置并在那里写字符。因为任务A 实际上并没有完成推进光标的工作，故任务B 的字符会覆盖任务A 的字符。这种情况发生的几率较低，但并不是不会发生。

因此，第42 行，在开始显示字符串之前，先禁止硬件中断；第54 行，只有在整个字符串完整地显示完毕之后，才开放硬件中断。这样，每个任务的字符串都能完整地显示。同时，因为本例程中有sti 指令，因此，在整个中断系统没有初始化完成之前，不能调用。

在硬件中断关闭期间，put_string 过程实际上是调用另一个近过程put_char 来逐个显示字符的。put_char 过程是从第62 行开始的。

第68～81 行的工作是取当前光标位置。取得的光标位置数值位于BX 寄存器中，要用于寻址显示缓冲区。因为访问显示缓冲区时用的是32 位寻址方式，故必须使用EBX 寄存器。第81 行，用and 指令清除EBX 寄存器的高16 位，仅保留低16 位（BX）。

第101 行，写字符到显示缓冲区。显示缓冲区的线性基地址是0x800B8000，缓冲区内的偏移量是由EBX 寄存器提供的，0x800B8000 是32 位立即数，故必须和EBX 寄存器搭配，而不能用BX 寄存器。还有，之所以显示缓冲区的线性基地址是0x800B8000，是因为物理内存的低端1MB被完整地映射到从0x80000000 开始的高端。因此，线性地址0x800B8000 会被处理器的页部件转换成物理地址0x000B8000。

同样的道理，第111～121 行，在做屏幕上滚的操作时，要传送的内容在4GB 段内的偏移量为0x800B80A0；目标位置在4GB 段内的偏移量为0x800B8000。

17.4 内核任务的创建

17.4.1 创建内核任务的TCB

回到第960～986 行，他往常一样，在屏幕上显示处理器的品牌信息，没有什么好说的。

第989～1007 行，在全局描述符表（GDT）中安装调用门，为用户任务提供系统服务。然后，通过调用门在屏幕上显示信息，以测试调用门的安装是否正确。

下面的工作是创建内核任务，也就是我们所说的程序管理器任务。内核任务需要一个任务控制块（TCB），毕竟它也要参与任务轮转。但是，该TCB 所需的内存不是动态分配的，而是一段静态的空间，是在内核程序编写的时候保留的。回到前面第519 行，在那里声明了标号core_tcb，并初始化了32 个为零的双字。TCB 不需要这么多空间，但多保留一些也没坏处。

第1010 行，首先设置内核任务的状态值为0xFFFF（忙）。实际上，当前正在执行的就是内核，从某种意义上来说就是内核任务，只不过没有办理将其TSS 描述符传送到任务寄存器TR 的手续而已。

内核占据着它自己的虚拟内存空间的高端，同时也映射到每个任务的虚拟内存空间的高端，具体的起始位置是线性地址0x80000000。从这里开始，前1MB（0x80000000～0x800FFFFF）已经被它自己用完了，实际可以继续分配的空间从线性地址0x80100000 开始。因此，第1011 行，在TCB 中设置这个可以分配的起始地址。

每个任务都可以有自己的LDT，如果没有也不要紧。第1013 行，设置内核任务的LDT 的初始界限值。就本章来看，这个值是用不上的，因为内核任务没有LDT。

对TCB 的初始化基本就是这些。第1015 行，将内核任务的TCB 追加到TCB 链表中，像从前一样，在TCB 链表中添加新的TCB 需要调用过程append_to_tcb_link，该过程位于第845 行，在本章已经做了修改，因此适合在平坦模型下工作。

当程序员不是一件容易的事，需要考虑的东西太多。访问和修改TCB 链原本不是多大的事情，可要是中断掺和进来，就得小心了。要知道，0x70 号实时时钟中断随时都在发生，那个中断处理过程也在不停地访问同一个TCB 链表。如果处理不当，很容易出现问题。请考虑一下，过程append_to_tcb_link 主要完成两件事：

① 遍历链表，找到最后一个TCB，修改它的“下一个TCB 线性地址”域，使它指向新的TCB；

② 清空新TCB 的“下一个TCB 线性地址”域，表明它是最后一个TCB。

假如现在已经完成了步骤1，新TCB 已经成为链表的最后一个节点。但是，在准备执行步骤2时，0x70 号中断发生了，该中断处理过程遍历链表。可想而知，因为新TCB 的“下一个TCB 线性地址”域还没有清零，所以，中断处理过程将无法找到链尾，而且会用那个非零的数作为地址，访问到它不该访问的区域（不存在的下一个TCB），处理器产生异常，程序很可能因此崩溃了！

因此，在过程append_to_tcb_link 的一开始，也就是第847 行，先用cli 指令屏蔽硬件中断。

和老版本相比，新版本的append_to_tcb_link 过程显然很简洁，毕竟它工作在平坦模型下。遍历链表找到最后一个TCB 的代码可以参考过程rtm_0x70_interrupt_handle，它们是相同的，不再赘述。

找到最后一个TCB 之后，第861 行，修改它的“下一个TCB 的线性地址”域，使其内容为新TCB 的线性地址；第862 行，将新TCB 的“下一个TCB 的线性地址”域清零。这两行很危险，在它们中间很容易发生中断。而一旦发生中断，麻烦就来了。cli 和sti 指令不必放在该过程的首尾，放到这两条指令的前后就行：

```
cli
mov [eax],ecx
mov dword [ecx],0x00000000   ;当前 TCB 指针域清零
sti
```

实际上，这是更合理的做法。记住，CLI 指令只在最有必要的时候使用。在一个正常的系统中，大家都很忙，都需要马不停蹄地投入运行，不要让无谓的中断屏蔽指令影响到每个程序的正常执行。

17.4.2 宏汇编技术

接下来是创建内核任务的TSS。对于一个任务来说，任务状态段（TSS）是必不可少的。为此，需要在内核的虚拟地址空间内分配内存。

第1018 行的作用是分配创建TSS 所需要内存空间：

```
alloc_core_linear
```

这是非常奇怪的，因为，它既不是处理器指令，也不像标号，居然还能单独存在。事实上，在汇编语言里，这是合法的，因为它是宏。

宏（Macro）是一种简化汇编语言程序编写的强大手段，绝大多数汇编语言编译器都支持宏，因此，这些汇编语言称为宏汇编。

宏并不是处理器指令，但它也不同于编译器提供的伪指令，专业地说，它是预处理指令。我们知道，编译器在编译源程序时，要多遍扫描，每次都完成不同的工作，这些都可以称为预处理，最后才开始将语句翻译成机器指令。要想说明宏是什么，下面是一个例子：

```
%define vrm(x)  0xb8000+x


[bits 32]
mov byte [vrm(0x02)],'h'
```

以上，%define 用来定义单行的宏，宏的名字叫vrm，带有一个参数x，参数要放在括号中。如果有多个参数，参数之间要用逗号分开。在宏的名字之后，要有空格，一个或者多个空格均可，然后，是一个表达式。从该表达式可以看出这个宏是用来做什么的，有什么意义。

宏的作用是可以代替复杂的表达式。在编译期间，编译器要先将宏展开，再编译成机器指令。因此，在编译期间，上面那条mov 指令将被展开为下面的形式：

```
mov byte [0xb8000+0x02],’h’
```

宏可以定义成任何形式，只要你觉得方便。因此，下面又是另一种定义宏和使用宏的例子：

```
%define vrm(x)  byte [0xb8000+x]

[bits 32]
mov vrm(ebx),'h'
```

在编译阶段，这最后一条mov 指令编译时，将先被展开成如下的形式：

```
mov byte [0xb8000+ebx],’h’
```

注意，宏定义不占用程序的地址空间，它只是一种简化程序编写过程的手段，仅在编译之前有用，在编译之后，宏就消失了。

除了单行的宏，多行的宏可能用处更大。定义多行的宏应当使用关键字%macro。多行宏的形式是

```
%macro  宏名字  参数个数
        … …        ;（宏的具体内容）
%endmacro
```

举个例子，假如有以下定义宏和使用宏的代码：

```
%macro  dostack  2

        push ebp
        mov ebp,esp
        sub esp,%1
        add dword [0x2000],%2
%endmacro

[bits 32]
dostack 8,0x55aa
```

注意，在定义宏的时候，宏名字后面只给出了参数个数，在宏体内使用参数时，第1 个参数是%1，第2 个参数是%2，依此类推。因此，以上代码编译时，最后一条语句将被展开成如下的形式：

```
push ebp
mov ebp,esp
sub esp,8
add dword [0x2000],0x55aa
```

回到代码清单17-2 中来。

宏alloc_core_linear 是在第12 行定义的，具有0 个参数，也就是没有参数。该宏的作用是在内核的虚拟地址空间上分配内存，并返回起始的线性地址。下一个可分配的线性地址在内核任务的TCB 中。因此，第13 行，从内核任务的TCB 中取得该地址。

为了简单起见，每次在内核的空间中分配内存时，不管需要多少，都固定地分配一个页（的大小）。因此，第14 行，将TCB 中的那个数值在原来的基础上增加4096（0x1000），这就是下一个可分配的线性地址。

第15 行，调用过程alloc_inst_a_page 以分配一个物理页，并用页的物理地址和它所对应的线性地址去修改当前内核任务的页目录表和页表。

当然，你可能怀疑在这里使用宏的必要性。事实上，这是有必要的。分配内存需要访问TCB，分配之后还要更新原来的数据，以形成下一个可分配的线性地址。很多时候，由于一时马虎，会忘了更新那个数，以至于下次分配的线性地址还和上一次相同，这就是致命的问题。一个可能的办法是将它定义成过程。问题是，事情很简单，用过程调用的方法来做代价太大。考虑再三，宏是最好的选择。

当然，不能滥用宏。否则，代码将既难阅读，又难维护。

为内核TSS 分配的线性基地址在EBX 寄存器中。第1021～1026 行，初始化TSS 中的静态部分，包括CR3 寄存器域（内核的页目录表物理地址）、LDT 域、TSS 反向链、I/O 位映射表偏移量等。

第1029～1033 行，创建内核任务TSS 的描述符。

第1034 行，将过程set_up_gdt_descriptor 返回的TSS 选择子保存到内核任务的TCB 中，将来在任务切换时要用到。

第1038 行，用内核任务的TSS 选择子加载任务寄存器TR。这将导致处理器将指定TSS 的线性基地址和段界限值加载到TR 寄存器的描述符

高速缓存器。至此，内核任务才名正言顺地成为一个合法的任务。

17.5　用户任务的创建

17.5.1　准备加载用户程序

和往常一样，接下来的工作是加载用户程序，创建用户任务。创建和撤销任务，是内核任务的职责所在。为此，需要首先创建用户任务的TCB。我们知道，为了能够访问和控制所有的任务，每个任务的TCB 都必须创建在内核的地址空间内。第1043 行，宏alloc_core_linear 用于创建用户任务的TCB。

第1045～1047 行，初始化用户任务的TCB，主要包括初始的LDT界限值，以及从哪个线性地址开始在用户任务的局部空间内分配内存（一般是0）。注意，任务的状态值是0x0000，即空闲。

第1049～1051 行，在当前栈中压入两个双字参数，并调用load_relocate_program 过程以加载和重定位用户程序。

过程load_relocate_program 是从第616 行开始的。

第620 行，保存栈指针寄存器ESP 的快照，为访问栈内的参数做准备。在第16 章，这条指令之前还有两个将段寄存器DS 和ES 压栈的指令：

```
pushad
push ds
push es
mov ebp,esp         ;为访问通过栈传递的参数做准备
```

而在本章中，内核和用户任务都工作在平坦模型下，也就不必考虑分段，段寄存器压栈的指令也没有了，变成了这样：

```
pushad
mov ebp,esp       ;为访问通过栈传递的参数做准备
```

正因为如此，栈的状态也和上一章有所不同。如图17-18 所示，第一个参数的位置是SS:EBP+40；第二个参数的位置是SS:EBP+36。

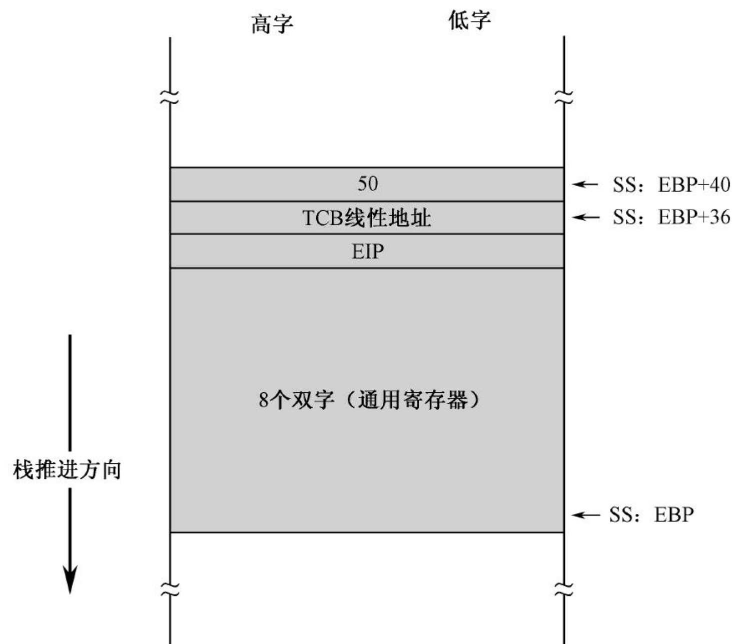


图17-18　执行mov ebp,esp 时的栈状态

这本来是不该成为一个问题的。想想看，要是当初把mov ebp,esp 指令放在pushad 指令之后，两个将段寄存器压栈的指令之前，就不至于在本章中做这样的修改工作了。

17.5.2　转换后援缓冲器的刷新

第625～631 行，清空当前页目录表的前半部分，为创建用户任务的页表目录项做准备。在第16 章里已经说清楚了，我们是借用内核的页目录来创建用户任务的页目录，毕竟，对于每一个任务来说，页目录表的前半部分对应着它的局部空间，后半部分对应着全局空间，内核用的是其页目录表的后半部分，前半部可以临时用来创建只属于任务自己的页目录项。

第633、634 行，重新加载一遍控制寄存器CR3（页目录表基地址寄存器PDBR）。显然，这是用CR3 原有的内容再次加载一遍，来回一样远，这有什么用呢？

开启页功能时，处理器的页部件要把线性地址转换成物理地址，而访问页目录表和页表是相当费时间的。因此，把页表项预先存放到处理器中，可以加快地址转换速度。为此，处理器专门构造了一个特殊的高速缓存装置，叫做转换后援缓冲器（Translation Lookaside Buffer，

TLB）。事实上，对该缓冲器的命名可谓五花八门，从“转换旁路缓冲器”、“转换后备缓冲区”到“快表”，不一而足。

如图17-19 所示，这是TLB 的结构。它分为两大部分，第一部分是标记，其内容为线性地址的高20 位；第二部分是页表数据，包括属性、访问权和页物理地址的高20 位。在分页模式下，当段部件发出一个线性地址时，处理器用线性地址的高20 位来查找TLB，如果找到匹配项（命中），则直接使用其数据部分的物理地址作为转换用的地址；如果检索不成功（不中），则处理器还得花时间访问内存中的页目录表和页表，找到那个页表项，然后将它填写到TLB 中，以备后用。TLB 容量不大，如果它装满了，则必须淘汰掉那些用得较少的项目。

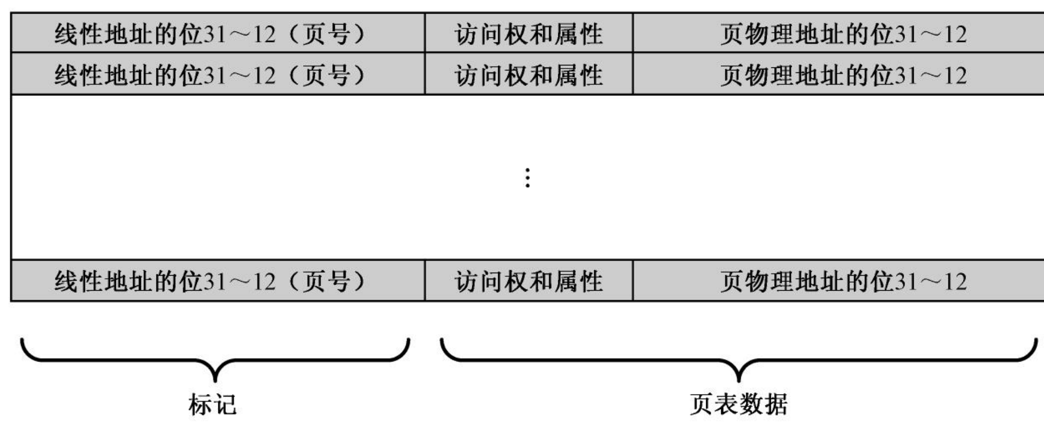


图17-19　转换后援缓冲器（TLB）的结构

TLB 中的属性位来自页表项，比如页表项中的D 位（脏位）等；访问权位来自页目录项和对应的页表项，比如RW 位和US 位，等等。问题是，就RW 位和US 位来说，页目录项和页表项都有这两位，以哪一个为准呢？在分页机制中，对页的访问控制按最严格的访问权执行。对于某个线性地址，如果其页目录项的RW 位是“0”而其页表项的RW 位是“1”，则按RW 位是“0”执行。也就是说，TLB 中的访问权，是页目录项和页表项中，对应访问权的逻辑与。

处理器仅仅缓存那些P 位是“1”的页表项，而且，TLB 的工作和CR3 寄存器的PCD 和PWT 位无关，不受这两位的影响。另外，对于页表项的修改不会同时反映到TLB 中。是的，这是很糟糕的，如果内存中的页表项已经修改，但TLB 中的对应条目还没有更新，那么，转换后的物理地址必定是错误的。

记得第一次在分页模式下写程序时，我遇到过这个问题。因为没有更新TLB，程序总是出错，总是产生异常（对这件事，网友周卫平应该是记得的）。在用bochs 软件单步跟踪程序的执行时，发现页目录项对应的页表项并不是我刚刚设置的。尽管你知道TLB，也知道它的原理，但是，很多时候，也许只有在花了几天工夫，熬了几夜之后，你才终于发现问题出在一个你明白，但却忽略了的地方。在第16 章里，只创建了一个用户任务，过程load_relocate_program 只被调用了一次，不会有什么问题。在本章里，起码创建了两个用户任务，如果不刷新TLB，是不行的。

TLB 是软件不可直接访问的，但却有其他办法来刷新它的内容（条目）。比如，将CR3 寄存器的内容读出，再原样写入，这样就会使得TLB 中的所有条目失效。当然，这是比较直接的做法。当任务切换时，因为要从新任务中的CR3 寄存器域加载页目录表基地址，也会隐式地导致TLB 中的所有条目无效。

注意，上述方法对于那些标记为全局（G 位为“1”）的页表项来说无效，不起作用。

17.5.3　用户任务的创建和初始化

从第637 行开始，到第760 行，这部分代码用于从硬盘上加载用户程序，并在LDT 内创建各个段的描述符。总体上，它们和第16 章相同，只有三点需要说明。

① 这段代码工作在平坦模式下，因此没有出现任何加载段寄存器和使用段超越前缀的指令。和用户任务有关的操作在虚拟地址空间的低端（0x00000000～0x7FFFFFFF）进行，和内核有关的操作在虚拟地址空间的高端（0x80000000～0xFFFFFFFF）进行。

② 在用户任务的局部地址空间分配内存时，用的是宏alloc_user_linear。该宏是在第18 行定义的。

③ 过程read_hard_disk_0 有所修改，主要是在首尾各增加了指令cli 和sti。在读硬盘时，应当屏蔽硬件中断，以防止对同一个硬盘控制器端口的交叉修改，这会产生很严重的问题。特别是在多任务环境下，当一个任务正在读硬盘时，会被另一个任务打断。如果另一个任务也访问硬盘，将破坏前一个任务对硬盘的操作状态。

第763～798 行，重定位U-SALT 表。这部分代码和第16 章相比，绝大多数是相同的，但也有差异。首先，去掉了一些指令，主要是那些加载段寄存器的指令，比如在第16 章中有如下指令：

```
mov eax,mem_0_4_gb_seg_sel     ;访问任务的 4GB 虚拟地址空间时用
mov es,eax

mov eax,core_data_seg_sel
mov ds,eax
```

在本章中，这些指令已经删除，因为内核和用户程序都工作在平坦模式下。

其次，同样的指令，在第16 章中的含义和本章不同。比如：

```
mov esi,salt
```

这条指令用于获取内核SALT 表（C-SALT）的地址。如图17-20（a）所示，在第16 章中，内核工作在多段模型，因此，ESI 寄存器中的内容是C-SALT 表在内核数据段内的偏移量。如图17-20（b）所示，在本章中，由于内核工作在平坦模式，因此，ESI 寄存器中的内容是C-SALT 表在整个4GB 段内的线性地址。

同第16 章相比，用户程序的基本结构没有变化。用户程序是从其虚拟地址空间的开始处（0x00000000）加载的，偏移量为0x0000000C 和0x00000008 的地方分别是U-SALT 表的条目数及其线性地址。第765、766 行用于获取这两个数值。

第801～807 行，创建LDT 描述符，并安装到GDT 中。

第809～824 行，用前面的工作成果来填写任务状态段（TSS）。

第827～832 行，创建TSS 描述符，并安装到GDT 中。

第836～838 行，当前页目录表只是借用的，它属于内核。用户任务必须有自己的页目录表。为此，申请一个空闲页作为用户任务的页目录表，并将当前页目录表的内容复制过去。

具体的复制工作是由过程create_copy_cur_pdir 来完成的，该过程是从第381 开始的。和第16章相比，多加了一条指令，即，第394 行的invlpg 指令（Invalidate TLB Entry）。

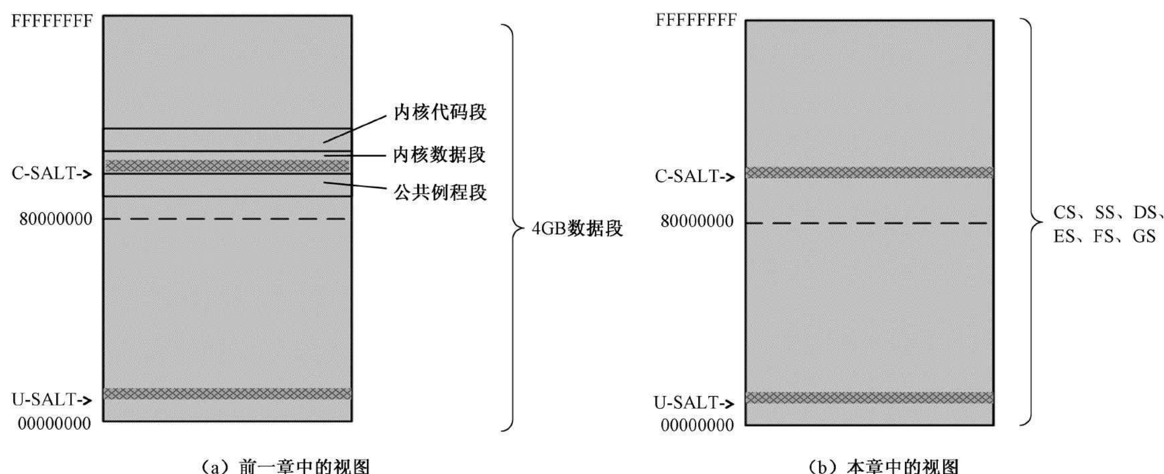


图17-20 U-SALT 和C-SALT 在两种内存模型下的布局视图

和刷新整个TLB 不同，invlpg 指令用于刷新TLB 中的单个条目。当然，要做到这一点，必须指定一个线性地址，处理器用给出的线性地址搜索TLB，找到那个条目，然后从内存中重新加载它。invlpg 指令的格式为

```
invlpg m
```

也就是说，该指令的操作数是一个内存地址。指令执行时，处理器首先确定该线性地址位于哪个页内，然后刷新相应的TLB 条目。你可能会问，为什么它不接受一个立即数，像这样：

```
invlpg 0xfffffff8
```

而非得是

```
invlpg [0xfffffff8]
```

我们知道，TLB 是一个附加的硬件机构，只有在处理器正常访问内存时才会导致它的填充和更新。因此，处理器用一个访问内存的操作来促使TLB 条目的更新会更方便。

0xFFFFFFF8 是当前（内核）页目录表内的倒数第2 个目录项，每次都用它来指向新任务的页目录表。当任务A 创建完毕后，它指向任务A 的页目录表；当任务B 创建时，它依然指向任务A的页目录表。虽然在第392 行改写了它，使它指向新任务的页目录表，但这个更改只在内存中

有效，还没有反映到TLB 中。如果不刷新TLB 中的这个条目，那么，后面的所有操作，都是针对前一个任务的页目录表进行的，这就麻烦了。是的，我遇到过这个问题。

因此，第394 行，用invlpg 指令来明确地刷新TLB 的对应条目。invlpg 是特权指令，在保护模式下执行时，当前特权级CPL 必须为0。该指令不影响任何标志位。

继续回到过程load_relocate_program，第842 行，弹出并废弃栈中的两个参数，将控制转移到调用者。

第1052、1053 行，将刚刚创建的那个任务的TCB 附加到TCB 链上。

注意，此刻，任务切换也跟着开始了！在此之前，TCB 链上只有一个状态为忙的内核任务，0x70 号中断虽然每秒产生一次，但不会有任务切换。随着第一个空闲任务的加入，当中断产生时，会切换到刚才创建的那个用户任务去执行。

用户程序非常简单，基本框架和第16 章一样。本章提供了两个用户程序，分别对应着代码清单17-3 和17-4，这两个程序执行时，分别对应着用户任务A 和用户任务B。请分别浏览这两个代码清单，你会发现它们基本一样，唯一的区别是它们在屏幕上显示的内容。任务A 的功能是不停地在屏幕上显示

```
User task A->;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

任务B 则显示的是

```
User task B->$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

第1056～1066 行，创建第二个用户任务。该任务对应着代码清单17-4，在虚拟硬盘上的起始逻辑扇区号为100。当该任务创建完毕，并加入TCB 链后，系统就会开始三个任务之间的切换。

第1068～1074 行是内核任务的主体，是反复执行的部分。它不做别的，就是在每次得到处理器控制权时，反复显示相同的信息“System core task running!”。理论上，作为任务的管理者，这里应该还有终止用户任务，并回收内存空间的代码。简单起见，这里予以省略，留给读者进行。

17.6　程序的编译和执行

分别编译本章提供的4 个代码清单，并生成相应的BIN 文件。

将文件c17_mbr.bin 写入虚拟硬盘的逻辑0 扇区，从逻辑1 扇区开始写入文件c17_core.bin，从逻辑50 扇区开始写入c17_1.bin；从逻辑100 扇区开始写入c17_2.bin。

启动虚拟机，正常情况下，程序的运行结果与图17-21 类似。

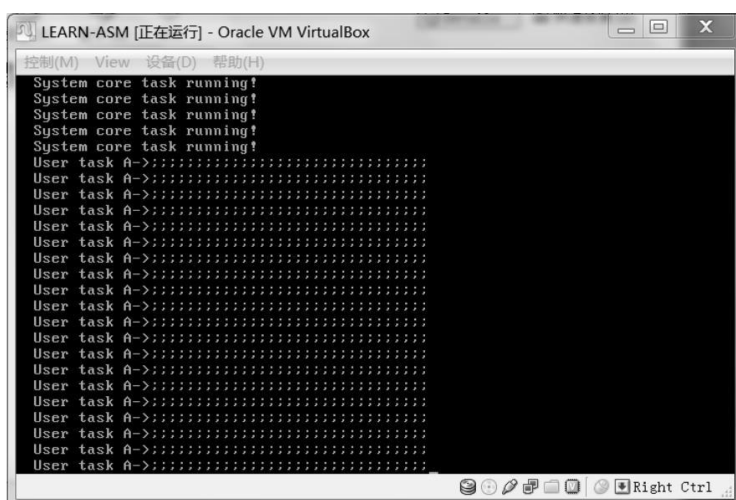


图17-21　本章程序的运行结果

本章习题

1．思考一下，在没有刷新TLB的情况下，为什么第二次调用load_relocate_program过程时会出现错误？

2．本章中的用户任务比较简单。结合你学过的数学知识，修改本章中的两个用户任务，使之完成比较复杂的数学运算，比如计算圆周率。

3．在平坦模型下，用户程序不能依靠段来实现自由浮动。如果可能的话，想办法解决这一问题。提示：一般来说，影响地址浮动的因素只和标号有关。

4．在流行的操作系统中，一般不用处理器固件实现任务切换，因为这种方法代价较高。如果可能的话，想办法采用软件的方法代替硬件实现任务切换。

附录Ⅰ　本书用到的x86指令及其页码

附录 II 本书用到的重要图表及其页码